A Separation Logic for Concurrent Randomized Programs

ANONYMOUS AUTHOR(S)

We present a concurrent separation logic with support for probabilistic reasoning. As part of our logic, we extend the idea of *coupling*, which underlies recent work on probabilistic relational logics, to the setting of programs with both probabilistic and non-deterministic choice. To demonstrate our logic, we verify a variant of a randomized concurrent counter algorithm and a two-level concurrent skip list. All of our results have been mechanized in Coq.

Additional Key Words and Phrases: separation logic, concurrency, probability

1 INTRODUCTION

Many concurrent algorithms use randomization to reduce contention and the need for coordination between threads. Roughly speaking, these algorithms are designed so that if each thread makes a local random choice, then on average the aggregate behavior of the whole system will have some good property.

For example, probabilistic skip lists [48] are known to work well in the concurrent setting [23, 28], because threads can independently insert nodes into the skip list without much synchronization. In contrast, traditional balanced tree structures are difficult to implement in a scalable way because re-balancing operations may require locking access to large parts of the tree.

However, concurrent randomized algorithms are difficult to write and reason about. Indeed, the use of just concurrency or randomness alone makes it hard to establish the correctness of an algorithm. For that reason, a number of program logics for reasoning about concurrent [19, 20, 24, 31, 33, 44, 46, 57] or randomized [7, 8, 34, 42, 49] programs have been developed.

But, to our knowledge, the only prior program logic designed for reasoning about programs that are both concurrent *and* randomized is the recent probabilistic rely-guarantee calculus developed by McIver et al. [39], which extends Jones' original rely-guarantee logic [31] with probabilistic constructs. However, this logic lacks many of the features of modern concurrency logics. For example, starting with Vafeiadis and Parkinson [57], many recent concurrency logics combine rely-guarantee style reasoning with some form of separation logic, which is useful for modular, local reasoning about fine-grained concurrent data structures.

In this paper we describe a logic which extends Iris [33], a state of the art concurrency logic, with support for probabilistic reasoning. By extending Iris, we ensure that our logic has the features needed to reason modularly about sophisticated concurrent algorithms. The key to our approach is several recent developments in probabilistic logic, concurrency logics, and denotational semantics. However, before we give an overview of how we build on this related work, let us describe a concurrent randomized algorithm, which will be our running example throughout this paper.

1.1 Example: Concurrent Approximate Counters

In many concurrent systems, threads need to keep counts of events. For example, in OS kernels, these counts can track performance statistics or reference counts. Somewhat surprisingly, Boyd-Wickizer et al. [14] have shown that maintaining such counts was a serious scalability bottleneck in a prior version of the Linux kernel. In many cases, however, there is no need for these counts to be *exactly* right: an estimate is good enough. Taking advantage of this, Dice et al. [17] created a scalable concurrent counter by adapting Morris's [43] *approximate counting* algorithm.

. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

```
incr l ≜
  let k = !l in
  let b = flip(1/2^k) in
  if b then l := k + 1
  else ();

read l ≜ let k = !l in 2^k − 1
```

(a) Sequential approximate counter.

```
incr l ≜
  let k = min(!l, MAX) in
  let b = flip(1/(k + 1)) in
  if b then (FAA(l, k + 1); ())
  else ();

read l ≜ !l
```

(c) Our unbiased concurrent counter.

```
incr l ≜
  let b = randbits(64) in
  incr_aux l b

incr_aux l b ≜
  let k = !l in
  if lsbZero(b, k) then
    if CAS(l, k, k + 1) then ()
    else incr_aux l b
  else ();
```

(b) Dice et al.'s concurrent counter.

Fig. 1. Approximate counting algorithms.

In order to understand Dice et al.'s concurrent version it is helpful to understand Morris's original work. Morris's motivation was to be able to count up to n using fewer than $O(\log_2(n))$ bits. His idea was that, rather than storing the exact count n, one could instead store something like $\log_2(n)$ rounded to the nearest integer. This would require only $O(\log_2 \log_2(n))$ bits, at the cost of the error introduced by rounding.

Of course, if we round the stored count, then when we need to increment the counter, we do not know how to update the rounded value correctly. Instead, Morris developed a randomized increment routine: if the counter currently stores the value k, then with probability $\frac{1}{2^k}$ we update the stored value to $k + 1$ and otherwise leave it unchanged. The code for this increment function is shown in Figure 1a, written in an ML-like pseudo code, where flip(p) is a command returning True with probability p and False otherwise. The read function loads the current value k of the counter and returns $2^k - 1$. Let C_n be the random variable giving the value stored in the counter after n calls of the increment function. One can show that $\mathbb{E}[2^{C_n} - 1] = n$. Thus, the expected value of the result returned by read is equal to the true number of increments, and so the counter is said to be an *unbiased* estimator. Flajolet [22] gave a very detailed analysis of this algorithm by showing that it is equivalent to a simple Markov chain, and proved that it indeed only requires $O(\log_2 \log_2(n))$ bits with high probability.

Although this space-saving property is interesting, the key aspect of the algorithm that makes it useful for concurrent counting is that, as the stored count gets very large, the probability that an increment needs to write to memory to update the count gets smaller and smaller. A simplified version of the concurrent algorithm proposed by Dice et al. [17] is shown in Figure 1b. The increment procedure starts by generating a large number of random bits. It then calls a recursive helper function incr_aux with the random bit vector b as an argument. This helper function reads the current value k stored in the counter and checks whether the first k bits of the bitvector are all 0 (performed in the code by the lsbZero(k, b) function). If not, the increment is over. If they are all zero, then because this occurs with probability $\frac{1}{2^k}$, the thread tries to atomically update the value stored in the counter from k to $k+1$ using a compare-and-swap (CAS) operation. If this operation succeeds, it means that no other thread has intervened and modified the counter, and so the increment is finished. If the swap fails, some other thread has modified the counter, so the incr_aux function is recursively called to try again. The read procedure is the same as for the sequential algorithm.

As the count gets larger, the probability that a thread will perform a CAS operation during the increment gets smaller, which is useful because these operations are slow. Dice et al. [17] show that this algorithm works quite well in practice, but do not give a formal argument for its correctness. Therefore, one might ask whether it is still guaranteed to give an unbiased estimate of the count. In fact, the answer is no: the scheduler can bias the count by ordering the compare-and-swap operations in a particular way.

In Figure 1c we present a new concurrent version that *is* statistically unbiased, yet retains the same good properties of low contention[1]. Our increment function reads the current value in the counter, then takes the minimum of that value and a parameter MAX. If the minimum value is k, then with probability $\frac{1}{k+1}$, it uses a fetch-and-add operation (FAA) to atomically add $k+1$ to the counter, otherwise it returns. In our version, the read function just returns the value in the counter. Like CAS instructions, FAA are expensive, so the reason the algorithm scales is that the probability that a FAA happens decreases as the counter value grows. The parameter MAX caps how small this probability gets, somewhat like generating only 64 random bits does in the beginning of Figure 1b.

How does one show that this algorithm is unbiased, as we have claimed? Informally, it is because in expectation, each increment adds 1 to the count, so the total expected value is equal to the number of increments. Moreover, because addition is associative and commutative, it does not matter if other threads modify the counter in between when a flip happens and the corresponding FAA occurs. However, it is challenging to make this argument formal. We might try to model the value of the counter as a family of Markov chains[2], as Flajolet did for the sequential algorithm. But this is unwieldy because the relevant state of the chain is not just the current value stored in l, but also the local state of each thread in the middle of an increment operation. Moreover, even if one could model the algorithm in this way, it is hard to justify the connection between the concrete implementation and this mathematical representation.

As we will see, the program logic we have developed makes this algorithm easy to verify.

1.2 Background from Recent Work

Our program logic is based on three ideas developed in recent work:

pRHL: probabilistic relational reasoning. In many program logics for reasoning about probabilistic programs, assertions in the logic either explicitly make statements about probabilities, or are interpreted as being true with some probability (*e.g.,* [7, 34, 42, 49], among others). Although effective, this non-standard semantics of assertions is hard to reconcile with the semantic models used in concurrency logics.

However, Barthe et al. have shown that reasoning about probabilistic programs can often be done *without* explicitly reasoning about probabilities in the assertions of the logic. Using their pRHL logic [5, 8, 9], one establishes a refinement between two randomized programs using proof rules that encode a special type of simulation relation. The only time that explicit probabilities

[1]However, it requires $O(\log_2(n))$ bits to store the count. Nevertheless, it uses less space than other alternatives for decreasing contention (*e.g.,* having each thread maintain its own local counter).

[2]Or rather, a *Markov decision process*, which accounts for the non-determinism of the ordering of operations.

arise is a special rule for points in the simulation when *both* programs take a randomized step. The soundness theorem for their logic says that derivations in the logic imply the existence of a *coupling*, a construct from probability theory that is often used to relate two probability distributions. (We describe couplings in detail in §2.5).

Iris: a "layered" concurrency logic. Modern concurrency logics are rather complex, making it hard to adapt them to incorporate probabilistic reasoning. Recent work, however, has sought to unify and simplify these logics [19, 33, 44].

In particular, Iris [32, 33, 36], a recent higher order concurrent separation logic, is composed of two layers: a "base logic" and a derived "program logic" which is encoded on top of the base logic. Crucially, most of the difficult semantic constructions are developed in the base logic. We are able to encode probabilistic relational reasoning *à la* pRHL in Iris by only modifying the second layer. We therefore get all of the results developed in the base logic "for free", and retain the expressive features of Iris.

Indexed valuations: a monadic encoding. Our logic, like pRHL, is designed for relational reasoning: it establishes a refinement between two programs. That means, if we want to prove a property about some program e, we first come up with some simpler *specification* program e', use the logic to establish a refinement connecting properties of e' to those of e, and then reason about e'.

Therefore, we need to complement our logic with a way to express the simpler program e' and suitable tools for reasoning about it. We write these specification programs using the monad of indexed valuations developed by Varacca and Winskel [60], which makes it possible to combine operations for both probabilistic and non-deterministic choice effects. This monad has a clean equational theory that makes it possible to reason about probabilistic properties of programs expressed using it.

1.3 Our Contributions

We make several contributions:

- We develop results for reasoning about computations expressed in the monadic encoding of Varacca and Winskel. Although prior work had used this monad, or similar ones combining both effects, for denotational semantics [26, 27, 41, 54, 60] and to reason about small program equivalences [25], we found it necessary to develop new ways to reason about this monad. In particular, we develop an (in)equational theory of orderings between computations, and rewriting rules for bounding expected values. In addition, we adapt the notion of couplings to this setting. Finally, Varacca and Winskel focused on finite probabilistic and non-deterministic choice, but their constructions can be generalized to support countable probabilistic choice and unbounded non-deterministic choice, which we have done.
- We extend Iris with support for probabilistic relational reasoning in the style of pRHL, which lets us establish refinements between concurrent programs and these monadic representations.
- Using our logic, we prove that the concurrent approximate counter algorithm introduced in §1.1 is unbiased.
- We also verify a fine-grained concurrent two-level skip list, and bound the expected number of comparisons performed when searching for a key.

All of the results in this paper, including the soundness of our logic and the examples, have been mechanized in Coq.

We start by describing Varacca and Winskel's [60] monad for probabilistic and non-deterministic choice, and our results for reasoning about computations expressed in it (§2). We then describe our program logic (§3). In §4 we give a detailed explanation of how we apply our logic to the approximate counter example. Then in §5 we give an overview of the skip list example. Finally, we discuss additional related work in §6.

2 MONADIC REPRESENTATION

A common approach to reasoning about effectful programs in a dependently typed proof assistant is to model effects using a suitable monad, M. Using this monad, one represents an effectful program that returns a value of type T as a term of type $M(T)$. Next, one usually proves a series of equational rules for simplifying terms of type $M(T)$, and other lemmas for reasoning about such terms. This approach has been used for reasoning about a number of effects, including: state [45, 51], non-termination [16], non-determinism [25], and probabilistic choice [3, 25, 47, 58].

2.1 Monads for non-determinism *or* probability.

Let us recall common monadic encodings for non-deterministic and probabilistic choice (separately). For non-determinism, we can choose $M_{\mathsf{N}}(T) = \{l : \mathsf{List}\ T \mid l \neq \emptyset\}$. That is, such computations are represented as pure terms that return a non-empty *list* of results, where each element of the list represents one of the different non-deterministic outcomes. We say two terms A and B of type $M_{\mathsf{N}}(T)$ are equivalent, written $A \equiv B$, if their sets of elements are the same: for all x, $x \in A \leftrightarrow x \in B$. In addition to the standard monadic operations (bind and return) we can define an operation for non-deterministic choice between two operations:

$$A \cup B \triangleq \mathsf{append}(A, B)$$

That is, we simply append the list of results for computation B to those of A. This operation satisfies a number of natural rules:

$$A \cup B \equiv B \cup A \qquad A \cup (B \cup C) \equiv (A \cup B) \cup C \qquad A \cup A \equiv A$$

These, in combination with the usual monad laws, can be used to prove that one non-deterministic computation is equivalent to another.

We can represent a probabilistic computation of type T as a list of pairs of values of type T along with probabilities that they occur, subject to the constraint that the sum of the probabilities is equal to 1:

$$M_{\mathsf{P}}(T) \triangleq \left\{ l : \mathsf{List}(T \times \mathbb{R}) \;\middle|\; \sum_{(x,p) \in l} p = 1 \right\}$$

Given a value v of type T and a probabilistic computation A of type $M_{\mathsf{P}}(T)$, we write $A|v$ for the sublist of A containing only elements whose first components are equal to v. The probability that a computation A returns v, written $A(v)$, is then:

$$A(v) \triangleq \sum_{(x,p) \in A|v} p$$

We say $A \equiv B$ if for all x, $A(x) = B(x)$.

Given such a computation A and a real number p, we write $p \cdot A$ for the list in which we multiply the second component of each element of A by p. Using this, we can define an operation which selects between a computation A with probability p and another computation B with probability $(1 - p)$:

$$A \oplus_p B \triangleq \mathsf{append}(p \cdot A, (1 - p) \cdot B)$$

This operation satisfies equational rules such as:

$$A \oplus_p B \equiv B \oplus_{1-p} A \qquad A \oplus_p A \equiv A$$

2.2 Combining effects

In order to reason about programs that use both probability and non-determinism, we would like some way to *combine* the monads we have just described. We might try to represent computations of type T combining both effects as terms of type $M_{\mathsf{N}}(M_{\mathsf{P}}(T))$, *i.e.,* non-empty lists of probability distributions.

But how do we define the monad operations for this combination? One way to derive the monad operations for the combination is to specify a *distributive law* [11]. However, Varacca and Winskel [60] have given a proof (based on an idea they attribute to Plotkin) that no distributive law exists between the monads we have described above. For our purposes, it is not necessary to understand the impossibility proof. What is important is that, based on their impossibility arguments, Varacca and Winskel observe that the following equational law, which holds in the probabilistic choice monad, is problematic if we want to have a distributive law:

$$A \oplus_p A \equiv A$$

At first this equivalence seems entirely natural: if in either case we choose A, then the probabilistic choice was irrelevant. However, when we later add in the effect of non-determinism, removing this law becomes more justifiable, since it allows us to account for the fact that subsequent non-determinism in the computation can be *resolved differently* on the basis of this seemingly irrelevant probabilistic choice.

Using this observation, Varacca and Winskel describe an alternative way of representing probabilistic choice, which they call the *indexed valuation monad*, in which this equivalence does not hold, and they then describe a distributive law between non-empty lists and these indexed valuations to obtain a monad combining both effects.

An indexed valuation of type T is a tuple $(I, \mathsf{ind}, \mathsf{val})$, where I is a finite set whose elements are called *indices*; ind is a function of type $I \to T$, and val is a function of type $I \to \mathbb{R}$ such that[3]:

$$\sum_{i \in I} \mathsf{val}(i) = 1$$

Informally, we can think of the indices as a set of "codes" or identifiers, the val function gives the probability of a particular index occurring, and ind maps these codes to elements of type T. Importantly, the ind function is not required to be injective, so that different codes can lead to the same observable result. We write $M_{\mathsf{I}}(T)$ for the type of indexed valuations of type T. The *support* of the valuation, notated support(val), is the set of indices i for which $\mathsf{val}(i) > 0$. We say $(I_1, \mathsf{ind}_1, \mathsf{val}_1) \equiv (I_2, \mathsf{ind}_2, \mathsf{val}_2)$ if there exists a bijection $h : \mathsf{support}(\mathsf{val}_1) \to \mathsf{support}(\mathsf{val}_2)$ such that for all $i \in \mathsf{support}(\mathsf{val}_1)$, $\mathsf{val}_1(i) = \mathsf{val}_2(h(i))$ and $\mathsf{ind}_1(i) = \mathsf{ind}_2(h(i))$. That is, the bijection can only "relabel" indices in a way that preserves their probabilities and what they decode to. Note that there is a map H which takes indexed valuations of type T to elements of $M_{\mathsf{P}}(T)$, defined by:

$$H(I, \mathsf{ind}, \mathsf{val}) = [(\mathsf{ind}(i_1), \mathsf{val}(i_1)), \dots, (\mathsf{ind}(i_n), \mathsf{val}(i_n))]$$
$$\text{where } I = \{i_1, \dots, i_n\}$$

It is clear that if $\mathbb{I}_1 \equiv \mathbb{I}_2$, then $H(\mathbb{I}_1) \equiv H(\mathbb{I}_2)$. However, the converse is not true because $\mathbb{I}_1$ and $\mathbb{I}_2$ could have supports with different cardinalities.

The probabilistic choice between two indexed valuations is defined by:

$$(I_1, \mathsf{ind}_1, \mathsf{val}_1) \oplus_p (I_2, \mathsf{ind}_2, \mathsf{val}_2) \triangleq (I_1 + I_2, \mathsf{val}', \mathsf{ind}')$$

[3] In fact, Varacca and Winskel first define a more general structure in which the sums of $\mathsf{val}(i)$ do not have to equal 1, and the set of indices does not have to be finite. After working out some of theory of these more general objects, they restrict to the subcategory satisfying these additional constraints.

$$\mathcal{I}_1 \oplus_p \mathcal{I}_2 \equiv \mathcal{I}_2 \oplus_{1-p} \mathcal{I}_1 \qquad \mathcal{I}_1 \oplus_1 \mathcal{I}_2 \equiv \mathcal{I}_1 \qquad \mathcal{I} \cup \mathcal{I} \equiv \mathcal{I} \qquad \mathcal{I}_1 \cup \mathcal{I}_2 \equiv \mathcal{I}_2 \cup \mathcal{I}_1$$

$$\mathcal{I}_1 \cup (\mathcal{I}_2 \cup \mathcal{I}_3) \equiv (\mathcal{I}_1 \cup \mathcal{I}_2) \cup \mathcal{I}_3 \qquad \mathcal{I}_1 \oplus_p (\mathcal{I}_2 \cup \mathcal{I}_3) \equiv (\mathcal{I}_1 \oplus_p \mathcal{I}_2) \cup (\mathcal{I}_1 \oplus_p \mathcal{I}_3)$$

$$x \leftarrow \mathcal{I}_1 \cup \mathcal{I}_2\,;\; F(x) \equiv (x \leftarrow \mathcal{I}_1\,;\; F(x)) \cup (x \leftarrow \mathcal{I}_2\,;\; F(x))$$

$$x \leftarrow \mathcal{I}_1 \oplus_p \mathcal{I}_2\,;\; F(x) \equiv (x \leftarrow \mathcal{I}_1\,;\; F(x)) \oplus_p (x \leftarrow \mathcal{I}_2\,;\; F(x))$$

Fig. 2. Equational laws for $M_\mathsf{N} \circ M_\mathsf{I}$ monad.

where:

$$\mathsf{ind}'(i) = \begin{cases} \mathsf{ind}_1(i') & \text{if } i = \mathsf{inl}(i') \\ \mathsf{ind}_2(i') & \text{if } i = \mathsf{inr}(i') \end{cases}$$

and

$$\mathsf{val}'(i) = \begin{cases} p \cdot \mathsf{val}_1(i') & \text{if } i = \mathsf{inl}(i') \\ (1-p) \cdot \mathsf{val}_2(i') & \text{if } i = \mathsf{inr}(i') \end{cases}$$

One can show that for all indexed valuations $\mathbb{I}_1$ and $\mathbb{I}_2$ and $0 \le p \le 1$, we have $\mathbb{I}_1 \oplus_p \mathbb{I}_2 \equiv \mathbb{I}_2 \oplus_{1-p} \mathbb{I}_1$.

However, unlike the original probabilistic choice monad we described before, $\mathbb{I} \oplus_p \mathbb{I} \not\equiv \mathbb{I}$, unless $p = 0$ or $p = 1$. The reason is that, when p is neither 0 nor 1, the support of $\mathbb{I} \oplus_p \mathbb{I}$ will have a larger cardinality than the support of $\mathbb{I}$, so there can be no bijection between them. Recall that we do *not* want this equivalence to hold, because it is the one that Varacca and Winskel [60] identified as problematic. With this obstruction removed, it is possible to define the monad operations on $M_\mathsf{N} \circ M_\mathsf{I}$. Given $\mathcal{I}_1$ and $\mathcal{I}_2$ of type $M_\mathsf{N}(M_\mathsf{I}(T))$, the probabilistic choice operation $\mathcal{I}_1 \oplus_p \mathcal{I}_2$ is defined by taking the pairwise probabilistic choice of each indexed valuation in the respective lists:

$$\mathcal{I}_1 \oplus_p \mathcal{I}_2 \equiv [\mathbb{I}_1 \oplus_p \mathbb{I}_2 \mid \mathbb{I}_1 \in \mathcal{I}_1, \mathbb{I}_2 \in \mathcal{I}_2]$$

while the non-deterministic choice is as before:

$$\mathcal{I}_1 \cup \mathcal{I}_2 = \mathsf{append}(\mathcal{I}_1, \mathcal{I}_2)$$

We say $\mathcal{I}_1 \equiv \mathcal{I}_2$ if for each $\mathbb{I}_1 \in \mathcal{I}_1$, there exists some $\mathbb{I}_2 \in \mathcal{I}_2$ such that $\mathbb{I}_1 \equiv \mathbb{I}_2$, and vice versa. The full definition of the bind operation is somewhat involved, as are the proofs of the monad laws, so we refer to Varacca and Winskel [60]. What is important is the equational properties that hold, of which a selection are shown in Figure 2 (the standard monad laws are omitted).

So far, by requiring the indices of the indexed valuations to be finite, we can only model sampling from finite distributions. Similarly, by only considering lists of indexed valuations we can only represent non-deterministic choice between a finite number of alternatives. However, all of the above can be generalized by allowing a *countable* number of indices and arbitrary non-empty sets of indexed valuations. This extra generality lets us model sampling from arbitrary discrete distributions. Throughout the rest of this paper and our mechanization, we use this more general definition.

2.3 Example: Modeling approximate counters

In Figure 3 we show how to model the approximate counter code from Figure 1c using this monad. The approxIncr computation first non-deterministically selects a number k up to MAX – this models the process of taking the minimum of the value in l and MAX in the code. The non-determinism

$$\begin{array}{l} \mathsf{approxIncr} \triangleq \\ \quad k \leftarrow \mathsf{ret}\ 0 \cup \cdots \\ \qquad\quad \cup\ \mathsf{ret}\ \mathsf{MAX}\,; \\ \quad \mathsf{ret}\ (k+1) \oplus_{\frac{1}{k+1}} \mathsf{ret}\ 0 \end{array} \qquad \begin{array}{l} \mathsf{approxN}\ 0\ l \triangleq \mathsf{ret}\ l \\ \mathsf{approxN}\ (n+1)\ l \triangleq \\ \quad k \leftarrow \mathsf{approxIncr}\,; \\ \quad \mathsf{approxN}\ n\ (l+k) \end{array}$$

Fig. 3. Monadic encoding of approximate counter algorithm from Figure 1c.

accounts for the fact that the value that will be read depends on what other threads do. The monadic encoding then makes a probabilistic choice, returning $k+1$ with probability $\frac{1}{k+1}$ and 0 otherwise, which represents the probabilistic choice that the code will make about whether to do the fetch-and-add.

Finally, the process of repeatedly incrementing the counter n times is modeled by approxN. The first argument n tracks the number of pending increments to perform, while the second argument l accumulates the sum of the values returned by the calls to approxIncr. Note that this model *does not* try to represent multiple threads in the middle of an increment each waiting to add its value to the shared count – rather, it is *as if* the actual calls to `incr` all happened atomically in sequential order, with the effects of concurrency captured by the non-determinism in the approxIncr computation.

Of course, we need to show that this model accurately captures the behavior of the code from Figure 1c – this is what the program logic we describe in §3 will do. First, however, we need to describe the new results we have developed to reason further about the monadic encoding itself.

2.4 Reasoning about Quantitative Properties

With what we have described so far, we can express computations with randomness and non-determinism and derive equivalences between them, but we do not yet have a way to talk about the standard concerns of probability theory (*e.g.*, expected values, variances, tail bounds).

Given an indexed valuation $\mathbb{I} = (I, \mathsf{ind}, \mathsf{val})$ of type T and a function $f : T \to \mathbb{R}$, we can define the *expected value* of f on $\mathbb{I}$ as:

$$\mathbb{E}_f[\mathbb{I}] \triangleq \sum_{i \in I} f(\mathsf{ind}(i)) \cdot \mathsf{val}(i)$$

(this coincides with the usual notion of expected value of a random variable if we interpret the indexed valuation as a distribution using the map H defined above). Since I may be a countable set (following the generalization we mentioned at the end of §2.2), the above series may not necessarily converge. We say that the expected value of f on $\mathbb{I}$ exists if the above series converges absolutely. Throughout this paper, when we mention expected values in rules and derivations, we will implicitly assume side conditions stating that all the relevant expected values exist.

If $\mathbb{I}_1 \equiv \mathbb{I}_2$, then for all f, $\mathbb{E}_f[\mathbb{I}_1] = \mathbb{E}_f[\mathbb{I}_2]$. Moreover, given a value t of type T, if we take f to be the *indicator function* that returns 1 if its input is equal to t and 0 otherwise, then $\mathbb{E}_f[\mathbb{I}]$ is equal to the probability that $\mathbb{I}$ yields the value t, so we can encode probabilities as expected values.

Since an $\mathcal{I}$ of type $M_{\mathsf{N}}(M_{\mathsf{I}}(T))$ is just a set of indexed valuations, we can apply $\mathbb{E}_f[-]$ to each $\mathbb{I} \in \mathcal{I}$ to get the set of expected values that can arise depending on how non-deterministic choices are resolved. Generally speaking, we will be interested in bounding the smallest or largest possible value that these expected values can take. We can define the *minimal* and *maximal* expected value of f on $\mathcal{I}$ as:

$$\mathbb{E}^{\mathsf{min}}_f[\mathcal{I}] \triangleq \inf_{\mathbb{I} \in \mathcal{I}} \mathbb{E}_f[\mathbb{I}] \qquad\qquad \mathbb{E}^{\mathsf{max}}_f[\mathcal{I}] \triangleq \sup_{\mathbb{I} \in \mathcal{I}} \mathbb{E}_f[\mathbb{I}]$$

$$\mathbb{E}^{\mathsf{min}}_f[\mathsf{ret}\ v] = f(v) \qquad \frac{k \geq 0}{\mathbb{E}^{\mathsf{min}}_{(\lambda x. k \cdot f(x)+c)}[\mathcal{I}] = k \cdot \mathbb{E}^{\mathsf{min}}_f[\mathcal{I}] + c}$$

$$\mathbb{E}^{\mathsf{min}}_f[\mathcal{I}_1 \oplus_p \mathcal{I}_2] = p \cdot \mathbb{E}^{\mathsf{min}}_f[\mathcal{I}_1] + (1-p) \cdot \mathbb{E}^{\mathsf{min}}_f[\mathcal{I}_2] \qquad \mathbb{E}^{\mathsf{min}}_{g \circ f}[\mathcal{I}] = \mathbb{E}^{\mathsf{min}}_g[x \leftarrow \mathcal{I}\ ;\ \mathsf{ret}\ f(x)]$$

$$\frac{\forall x.\, k_1 \leq \mathbb{E}^{\mathsf{min}}_f[F(x)] \leq k_2}{k_1 \leq \mathbb{E}^{\mathsf{min}}_f[x \leftarrow \mathcal{I}_1\ ;\ F(x)] \leq k_2}$$

Fig. 4. Selection of rules for calculating extrema of expected values (analogous rules for $\mathbb{E}^{\mathsf{max}}_f[-]$ omitted).

We say that these extrema exist if for all $\mathbb{I} \in \mathcal{I}$, $\mathbb{E}_f[\mathbb{I}]$ exists. Since $\mathcal{I}$ may be an infinite set, $\mathbb{E}^{\mathsf{min}}_f[\mathcal{I}]$ and $\mathbb{E}^{\mathsf{max}}_f[\mathcal{I}]$ can be $-\infty$ and $+\infty$ respectively. Rules for calculating these values are given in Figure 4. As before, we implicitly assume that all of the stated extrema exist and are finite.

To help reason about these extrema, we introduce a partial order on terms of type $M_{\mathsf{N}}(M_{\mathsf{I}}(T))$: We say $\mathcal{I}_1 \subseteq \mathcal{I}_2$ if for each $\mathbb{I}_1 \in \mathcal{I}_1$, there exists some $\mathbb{I}_2 \in \mathcal{I}_2$ such that $\mathbb{I}_1 \equiv \mathbb{I}_2$. If $\mathcal{I}_1 \subseteq \mathcal{I}_2$ then $\mathbb{E}^{\mathsf{max}}_f[\mathcal{I}_1] \leq \mathbb{E}^{\mathsf{max}}_f[\mathcal{I}_2]$ and $\mathbb{E}^{\mathsf{min}}_f[\mathcal{I}_2] \leq \mathbb{E}^{\mathsf{min}}_f[\mathcal{I}_1]$. Thus, we can bound $\mathcal{I}_1$'s extrema by first finding some $\mathcal{I}_2$ such that $\mathcal{I}_1 \subseteq \mathcal{I}_2$, and then bounding the latter's extrema.

Although we have omitted side-conditions on the existence of expected values and extrema here, they must be dealt with in our mechanized proofs in Coq, which can be somewhat tedious. One way to discharge these side conditions is to show that the functions we are computing expected values of are suitably bounded. We first define the *support* of $\mathbb{I}$ as the set of all values that occur with non-zero probability:

$$\mathsf{supp}(\mathbb{I}) \triangleq \{v \mid \exists i \in \mathbb{I}.\, \mathsf{ind}(i) = v \wedge \mathsf{val}(i) > 0\}$$

The support of a set of indexed valuations, $\mathcal{I}$ is then the union of their supports:

$$\mathsf{supp}(\mathcal{I}) \triangleq \bigcup_{\mathbb{I} \in \mathcal{I}} \mathsf{supp}(\mathbb{I})$$

We say that f is bounded on the support of $\mathcal{I}$ if there exists some c such that $|f(v)| \leq c$ for all $c \in \mathsf{supp}(\mathcal{I})$. If this holds, then $\mathbb{E}^{\mathsf{min}}_f[\mathcal{I}]$ and $\mathbb{E}^{\mathsf{max}}_f[\mathcal{I}]$ exist and are finite.

Using the above rules, we can show that $\mathbb{E}^{\mathsf{min}}_{\mathsf{id}}[\mathsf{approxN}\ n\ 0] = \mathbb{E}^{\mathsf{max}}_{\mathsf{id}}[\mathsf{approxN}\ n\ 0] = n$, which implies that no matter how the non-determinism is resolved in our model of the counter, the expected value of the result will be the number of increments. Let us just consider the case for the minimum, since the maximum is the same. The proof proceeds by induction on n, after first strengthening the induction hypothesis to the claim that $\mathbb{E}^{\mathsf{min}}_{\mathsf{id}}[\mathsf{approxN}\ n\ l] = n + l$. The key step of the proof is to show that $\mathbb{E}^{\mathsf{min}}_{\mathsf{id}}[\mathsf{approxIncr}] = 1$, *i.e.,* each increment contributes 1 to the expected value. From the last rule in Figure 4, it suffices to show that whatever value of k is non-deterministically selected, the resulting expected value will be 1. We have that for all k:

$$\begin{aligned} &\mathbb{E}^{\mathsf{min}}_{\mathsf{id}}\left[\mathsf{ret}\ (k+1) \oplus_{\frac{1}{k+1}} \mathsf{ret}\ 0\right] \\ &= \left(\frac{1}{k+1}\right) \cdot (k+1) + \left(1 - \frac{1}{k+1}\right) \cdot 0 \\ &= 1 \end{aligned}$$

2.5 Nondeterministic Couplings

Up to this point, we have described rules for the $\equiv$ equivalence relation and $\subseteq$ ordering on $M_{\mathsf{N}}(M_{\mathsf{I}}(T))$, and have shown how these can be used to reason about monadic computations. However, these relations are too fine to apply in many cases.

In contrast, in classical probability theory, there is a more flexible relation called a *coupling* [38] between two probability distributions. Recent work by Barthe et al. [4, 5, 9] has shown that this notion is fundamental for relational reasoning in probabilistic program logics. Given two distributions $A : M_{\mathsf{P}}(T_A)$ and $B : M_{\mathsf{P}}(T_B)$, a coupling between A and B is a distribution $C : M_{\mathsf{P}}(T_A \times T_B)$ such that:

(1) $\forall x : T_A.\, A(x) = \sum_y C(x,y)$
(2) $\forall y : T_B.\, B(y) = \sum_x C(x,y)$

That is, C is a joint distribution whose marginals equal A and B. These two conditions are equivalent to requiring that:

(1) $A \equiv ((x,y) \leftarrow C;\ \mathsf{ret}\ x)$
(2) $B \equiv ((x,y) \leftarrow C;\ \mathsf{ret}\ y)$

Given a predicate $P : A \times B \longrightarrow \mathsf{Prop}$, we say that C is a P-coupling, if, in addition to the above, we have:

$$\forall x, y.\, C(x,y) > 0 \longrightarrow P(x,y)$$

i.e., all pairs (x,y) in the support of the distribution C satisfy P. The existence of a P-coupling can tell us important things about the two distributions. For example, if $P(x,y) = (x = y)$, then the existence of a P-coupling tells us the two distributions are equivalent. Moreover, there are rules for systematically constructing couplings between distributions. We will explain some of these rules once we have described how to adapt couplings to the monad $M_{\mathsf{N}} \circ M_{\mathsf{I}}$.

First, using the monadic formulation of the coupling conditions, it is straightforward to define an analogous idea for M_{I}: Given $\mathbb{I}_1 : M_{\mathsf{I}}(T_1)$ and $\mathbb{I}_2 : M_{\mathsf{I}}(T_2)$, a coupling between $\mathbb{I}_1$ and $\mathbb{I}_2$ is an $\mathbb{I} : M_{\mathsf{I}}(T_1 \times T_2)$ such that:

(1) $\mathbb{I}_1 \equiv ((x,y) \leftarrow \mathbb{I};\ \mathsf{ret}\ x)$
(2) $\mathbb{I}_2 \equiv ((x,y) \leftarrow \mathbb{I};\ \mathsf{ret}\ y)$

and $\mathbb{I} = (I, \mathsf{ind}, \mathsf{val})$ is a P-coupling if for all i such that $\mathsf{val}(i) > 0$, $P(\mathsf{ind}(i))$ holds. As before, if P is the equality predicate, then the existence of a P-coupling between $\mathbb{I}_1$ and $\mathbb{I}_2$ implies $\mathbb{I}_1 \equiv \mathbb{I}_2$.

We can lift this to a relation between a single indexed valuation $\mathbb{I}$ and a set of indexed valuations $\mathcal{I}$: We say[4] there is a *non-deterministic P-coupling* between $\mathbb{I}$ and $\mathcal{I}$ if there exists some $\mathbb{I}' \in \mathcal{I}$ and a P-coupling between $\mathbb{I}$ and $\mathbb{I}'$. We write $\mathbb{I} \sim \mathcal{I} : P$ to denote the existence of such a coupling.

In this case, if P is the equality relation, then this means $\mathbb{I}$ is equivalent to some $\mathbb{I}'$ in $\mathcal{I}$, hence to bound the range of an expected value $\mathbb{E}_f[\mathbb{I}]$, it suffices to bound the extrema $\mathbb{E}_f^{\mathsf{min}}[\mathcal{I}]$ and $\mathbb{E}_f^{\mathsf{max}}[\mathcal{I}]$. More generally, we have the following:

Theorem 2.1. *Let g be bounded on supp($\mathcal{I}$) and let $P(x,y) = (f(x) = g(y))$. If $\mathbb{I} \sim \mathcal{I} : P$, then $\mathbb{E}_f[\mathbb{I}]$ exists and*

$$\mathbb{E}_g^{min}[\mathcal{I}] \leq \mathbb{E}_f[\mathbb{I}] \leq \mathbb{E}_g^{max}[\mathcal{I}]$$

Rules for constructing these couplings are shown in Figure 5. If we interpret the P in $\mathbb{I} \sim \mathcal{I} : P$ as a kind of "post-condition" for the execution of the computations $\mathbb{I}$ and $\mathcal{I}$, then these coupling

[4] Barthe et al. [4] use "non-deterministic coupling" to refer to a particular kind of coupling which is unrelated to adversarial non-deterministic choice.

$$\textsc{Ret}\ \frac{P(a,b)}{\mathsf{ret}\ a \sim \mathsf{ret}\ b : P} \qquad \textsc{Equiv}\ \frac{\mathbb{I} \equiv \mathbb{I}' \qquad \mathcal{I} \subseteq \mathcal{I}' \qquad \mathbb{I} \sim \mathcal{I} : P}{\mathbb{I}' \sim \mathcal{I}' : P} \qquad \textsc{Conseq}\ \frac{\mathbb{I} \sim \mathcal{I} : P \qquad \forall x, y.\, P(x,y) \to P'(x,y)}{\mathbb{I} \sim \mathcal{I} : P'}$$

$$\textsc{Bind}\ \frac{\mathbb{I} \sim \mathcal{I} : P \qquad \forall x, y.\, P(x,y) \to F(x) \sim F'(y) : Q}{(x \leftarrow \mathbb{I};\ F(x)) \sim (y \leftarrow \mathcal{I}\ ;\ F'(y)) : Q} \qquad \textsc{P-Choice}\ \frac{\mathbb{I} \sim \mathcal{I} : P \qquad \mathbb{I}' \sim \mathcal{I}' : P}{\mathbb{I} \oplus_p \mathbb{I}' \sim \mathcal{I} \oplus_p \mathcal{I}' : P}$$

Fig. 5. Rules for constructing non-deterministic couplings.

rules have the structure of a Hoare-like relational logic [12], as in the work of Barthe et al. [4]: *e.g.*, the rule Bind is analogous to the usual sequencing rule in Hoare logic.

The rule P-Choice lets us couple probabilistic choices $\mathbb{I} \oplus_p \mathbb{I}'$ and $\mathcal{I} \oplus_p \mathcal{I}'$ with post-condition P by coupling $\mathbb{I}$ to $\mathcal{I}$ and $\mathbb{I}'$ to $\mathcal{I}'$. This is somewhat surprising: we get to reason about these two probabilistic choices as if they both chose the left alternative or both chose the right alternative, rather than considering the full set of four combinations. This counter-intuitive rule is quite useful, as demonstrated in the many examples given in the work of Barthe et al. We will see an example of its use in §4.

3 PROGRAM LOGIC

We now describe the program logic we have developed for proving that a program is modeled by the monadic specifications from the previous section.

3.1 Program Semantics

Our logic is parameterized by a generic probabilistic concurrent language. However, for concreteness, we instantiate it with the ML-like language used in the examples from §1. Figure 6 gives the syntax and semantics of this language. We omit the standard rules for things like tuples, recursive functions, and references. The per-thread reduction relation $e; \sigma \xrightarrow{p} e'; \sigma'$ is annotated with a probability p that the transition takes place. We say e is atomic, written atomic(e), if e reduces to a value in a single step.

The flip(n_1, n_2) command takes two integers as arguments and simulates a biased coin flip: it transitions to True with probability $\frac{n_1}{n_2}$ and False with probability $1 - \frac{n_1}{n_2}$. (In the introduction we somewhat informally wrote flip(n_1/n_2) as if the language had rational numbers as a primitive). There is a side condition to ensure that $\frac{n_1}{n_2}$ actually corresponds to a valid probability. Other than this command, the per-thread transition system for this language is deterministic. The generic framework for our logic allows us to extend this language with other probabilistic commands, so long as they only sample from discrete probability distributions.

This per-thread reduction relation is then lifted to a concurrent transition system. A *configuration* ρ is a pair consisting of a list of expressions (representing a pool of threads) and a state σ. We say $\rho \xrightarrow[i]{p} \rho'$ when the i^{th} thread of ρ transitions with probability p leading to a new configuration ρ'. The fork$\{e_f\}$ command adds a new thread e_f to the pool.

A *scheduler* decides which thread will get to step at each point in an execution. We model a scheduler as a function φ of type $\mathit{Trace} \to \mathit{Option}\ \mathbb{N}$, where a trace is a non-empty list of configurations representing a partial execution. The scheduler is permitted to inspect the entire history and complete state of the program when deciding which thread gets to go next. Of course, a real implementation of a scheduler does not actually do this, but by conservatively considering

Syntax:

$$
\begin{array}{llll}
\mathit{Val} & v & ::= & \lambda x.\, e_1 \mid (v_1, v_2) \mid () \mid n \mid b \mid \ \ldots \\
\mathit{Expr} & e & ::= & x \mid v \mid e_1\, e_2 \mid \mathsf{fork}\{e\} \mid \mathsf{flip}(e_1, e_2) \mid \ \ldots \\
\mathit{Eval\ Ctx} & K & ::= & [] \mid K\, e \mid V\, K \mid \mathsf{flip}(K, e) \mid \mathsf{flip}(v, K) \mid \ \ldots \\
\mathit{State} & \sigma & \in & \mathbb{N} \rightharpoonup \mathit{Val} \\
\mathit{Config} & \rho & \in & \{l : \mathit{List\ Expr} \mid l \neq \emptyset\} \times \mathit{State} \\
\mathit{Trace} & T & \in & \{l : \mathit{List\ Config} \mid l \neq \emptyset\} \\
\mathit{Scheduler} & \varphi & \in & \mathit{Trace} \rightarrow \mathit{Option}\, \mathbb{N}
\end{array}
$$

Per-Thread Reduction: $e; \sigma \xrightarrow{p} e'; \sigma'$

Flip-True
$$\frac{0 \leq \dfrac{n_1}{n_2} \leq 1}{\mathsf{flip}(n_1, n_2); \sigma \xrightarrow{\frac{n_1}{n_2}} \mathsf{True}; \sigma}$$

Flip-False
$$\frac{0 \leq \dfrac{n_1}{n_2} \leq 1}{\mathsf{flip}(n_1, n_2); \sigma \xrightarrow{1-\frac{n_1}{n_2}} \mathsf{False}; \sigma}$$

(Standard rules omitted.)

Concurrent Semantics: $\rho \xrightarrow[i]{p} \rho'$

$$\frac{e_i; \sigma \xrightarrow{p} e_i'; \sigma'}{[\ldots, K[e_i], \ldots]; \sigma \xrightarrow[i]{p} [\ldots, K[e_i'], \ldots]; \sigma'}$$

$$[e_1, \ldots, e_{i-1}, K[\mathsf{fork}\{e_{\mathrm{f}}\}], \ldots]; \sigma \xrightarrow[i]{1} [e_1, \ldots, e_{i-1}, K[()], \ldots, e_{\mathrm{f}}]; \sigma$$

Trace Semantics: $T \xrightarrow[\varphi]{p} T'$

$$\frac{\varphi(T, \rho) = \mathsf{Some(i)} \qquad \rho \xrightarrow[i]{p} \rho'}{T, \rho \xrightarrow[\varphi]{p} T, \rho, \rho'} \qquad \frac{\varphi(T, \rho) = \mathsf{None}}{T, \rho \xrightarrow[\varphi]{1} T, \rho, \rho} \qquad \frac{\varphi(T, \rho) = \mathsf{Some(i)} \qquad \neg(\exists \rho', p.\, \rho \xrightarrow[i]{p} \rho')}{T, \rho \xrightarrow[\varphi]{1} T, \rho, \rho}$$

Fig. 6. Syntax and semantics of concurrent language.

this strong class of *adversarial* schedulers, results we prove will also hold for realistic schedulers. We write $T \xrightarrow[\varphi]{p} T'$ to indicate that the thread selected by $\varphi(T)$ steps with probability p to a new configuration which is appended to T to obtain T'. We write $\mathsf{curr}(T)$ for the last configuration in a trace. We permit a scheduler to return None, and if this happens, or if the scheduler returns a thread number which cannot take a step, the system takes a "stutter" step and the current configuration is repeated again at the end of the trace. We say φ is well-formed if, whenever $\varphi(T) = \mathsf{Some}(i)$, then $\mathsf{curr}(T)$ is of the form $([e_1, \ldots, e_n], \sigma)$, such that e_i, σ can take a step. That is, a well-formed scheduler always selects threads that can take steps. We say T reduces to T' in n steps under φ if:

$$T \xrightarrow[\varphi]{p_1} \ldots \xrightarrow[\varphi]{p_n} T'$$

for some $p_1, \ldots, p_n$ where each $p_i > 0$. A configuration ρ has terminated if the first thread in the pool is a value. We say that T is terminating in at most n steps under φ, if for all T' which T reduces to under φ for $n' \geq n$ steps, $\mathsf{curr}(T')$ has terminated.

We now want to interpret this reduction relation as defining a distribution on program executions. However, in general, this would require measure theoretic probability to handle properly: even though our language only features sampling from countable discrete distributions, the set of all executions of a program is uncountable if the program does not necessarily terminate[5]. However, if we restrict consideration to programs that terminate in a bounded number of steps, we can avoid these issues. Since most concurrency logics only handle partial correctness specifications anyway, this does not lead to much further loss of generality.

With this restriction in place, we can interpret program executions as indexed valuations (as we explained in §2, indexed valuations can be interpreted as probability distributions, and vice versa). Given a scheduler φ and a trace T, we first convert the trace step relation to an indexed valuation. Since the set of traces T' which T can step to under φ is countable, we can take the set of indices I to be any set in bijection with this set of traces. Take ind to be this bijection, and set $\mathsf{val}(i)$ equal to the probability p such that $T \xrightarrow[\varphi]{p} \mathsf{ind}(i)$. We refer to the resulting indexed valuation $(I, \mathsf{ind}, \mathsf{val})$ as $\mathsf{tstep}_\varphi(T)$. For each n, we define the indexed valuation $\mathsf{resStep}^n_\varphi(T)$ recursively by:

$$\begin{array}{lcl} \mathsf{resStep}^0_\varphi(T) & \triangleq & \mathsf{match}\ \mathsf{curr}(T)\ \mathsf{with} \\ & & \quad ([e_1, \ldots], \sigma) \Rightarrow \mathsf{ret}\ e_1 \\ & & \mathsf{end} \\ \mathsf{resStep}^{n+1}_\varphi(T) & \triangleq & T' \leftarrow \mathsf{tstep}_\varphi(T)\,;\ \mathsf{resStep}^n_\varphi(T') \end{array}$$

This corresponds to stepping the trace n times and returning the first thread from the final configuration of the resulting trace. We regard the "return value" of a concurrent program to be the value that the first thread evaluates to, so in the event that the program terminates in n steps under the scheduler, $\mathsf{resStep}^n_\varphi(T)$ gives this return value.

3.2 Background on Iris

As we have mentioned, our program logic is an extension of Iris, a recent concurrency logic with many expressive features. For reasons of space, we cannot explain all of Iris. We refer the reader to the Iris papers and manual [32, 33, 36, 53] for a full account. Instead, we will just describe some essential aspects needed to understand our extensions and examples.

Figure 7 shows the basic concurrent separation logic rules of Iris. (Treat the $\Rrightarrow$ connective as just a kind of implication for now, we explain its use below.) These rules are used to establish triples of the form:

$$\{P\}\ e\ \{x.\, Q\}$$

which imply that if e is executed in a state that initially satisfies P, and it terminates with value v, then the terminating state will satisfy $[v/x]Q$. Furthermore, at no point will e go wrong and reach a stuck state, and neither will any of the threads forked by e during its execution. This is a partial correctness property: the post-condition only holds under executions where e terminates.

The fundamental idea of separation logic [50] is the separating conjunction $P * Q$, which says that the program heap can be split into two disjoint pieces satisfying P and Q respectively. Thus,

[5] A more denotational alternative, based on an approach due to Kozen [35], is to interpret programs as monotone maps on sub-distributions of states. Then recursive commands are interpreted as least fixed points. However, since the original soundness proof of Iris is given in terms of a language with an operational semantics, we found it easier to use the semantics we describe in this section.

ML-ALLOC
$$\{\mathsf{True}\}\ \mathsf{ref}\ v\ \{x.\, x \hookrightarrow v\}$$

ML-LOAD
$$\{l \hookrightarrow v\}\ !l\ \{x.\, x = v \land l \hookrightarrow v\}$$

ML-STORE
$$\{l \hookrightarrow v\}\ l := w\ \{l \hookrightarrow w\}$$

ML-FAA
$$\{l \hookrightarrow n\}\ \mathsf{FAA}(l, k)\ \{x.\, x = n \land l \hookrightarrow n + k\}$$

ML-FORK
$$\frac{P \Rrightarrow Q_0 * Q_1 \qquad \{Q_0\}\ e\ \{\mathsf{True}\} \qquad \{Q_1\}\ e'\ \{R\}}{\{P\}\ \mathsf{fork}\{e\}; e'\ \{R\}}$$

HT-FRAME
$$\frac{\{P\}\ e\ \{v.\, Q\}}{\{P * R\}\ e\ \{v.\, Q * R\}}$$

HT-CSQ
$$\frac{P \Rrightarrow P' \qquad \{P'\}\ e\ \{v.\, Q'\} \qquad \forall v.\, Q' \Rrightarrow Q}{\{P\}\ e\ \{v.\, Q\}}$$

Fig. 7. Selection of rules from Iris.

CountGeq
$$\boxed{\bullet\, n}^{\gamma} * \boxed{\circ\, (q, n')}^{\gamma} \Rightarrow n \geq n'$$

CountEq
$$\boxed{\bullet\, n}^{\gamma} * \boxed{\circ\, (1, n')}^{\gamma} \Rightarrow n = n'$$

CountPerm
$$\boxed{\circ\, (q, n)}^{\gamma} * \boxed{\circ\, (q', n')}^{\gamma} \Rightarrow q + q' \leq 1$$

CountSep
$$\boxed{\circ\, (q, n)}^{\gamma} * \boxed{\circ\, (q', n')}^{\gamma} \Leftrightarrow \boxed{\circ\, (q + q', n + n')}^{\gamma}$$

CountAlloc
$$\mathsf{True} \Rrightarrow \exists \gamma.\, \boxed{\bullet\, n}^{\gamma} * \boxed{\circ\, (1, n)}^{\gamma}$$

CountUpd
$$\boxed{\bullet\, (n + k)}^{\gamma} * \boxed{\circ\, (q, n)}^{\gamma} \Rrightarrow \boxed{\bullet\, (n' + k)}^{\gamma} * \boxed{\circ\, (q, n')}^{\gamma}$$

Fig. 8. Counter resource rules.

assertions are interpreted as claims of *ownership of resources*, where a "resource" is just a fragment of the heap. In particular, the $l \hookrightarrow v$ assertion claims ownership of a part of the heap containing the location l, and moreover says that l maps to the value v. Ownership of this resource licenses a thread to access and modify this location (see ML-LOAD and ML-STORE in Figure 7).

Like other recent concurrency logics [2, 19, 44], Iris lets users of the logic extend the notion of resource beyond just heap fragments. These user-defined resources let us model the complex protocols that govern how threads access shared state in a concurrent system. Instead of describing the machinery that makes this work, let us give an example of one of these user-defined resources: the abstract "counter" resource. There are two types of these counter resources, represented by the following assertions:

$$\boxed{\bullet\, n}^{\gamma} \qquad \text{and} \qquad \boxed{\circ\, (q, n')}^{\gamma}$$

where n and n' are natural numbers, $0 < q \leq 1$ is a rational number, and γ is an abstract name assigned to a particular counter. The $\bullet\, n$ resource represents a shared counter that contains the value n. If we think of such a counter as being composed of n "units", then the resource $\circ\, (q, n')$ represents a "stake" or ownership of n' of the units in the global counter. The parameter q is a fractional permission [15] that lets us track how many threads have such a stake; when $q = 1$, this represents full ownership, so no other threads have a stake[6].

Rules for using these assertions are given in Figure 8. The rules CountGeq and CountEq let us conclude that the global counter value must be at least as big as any stake's value; and when a stake's q value is 1, we furthermore know that the counter and the stake value are the same. The

[6]Note that the q is *not* the fraction of the global counter value represented by the stake's value.

$$\textsc{inv-alloc}\quad P \Rrightarrow \exists \iota.\, \boxed{P}^{\iota} \qquad\qquad \textsc{inv-dup}\quad \boxed{P}^{\iota} \Rightarrow \boxed{P}^{\iota} * \boxed{P}^{\iota}$$

$$\textsc{inv-open}\quad \frac{\text{(additional side conditions omitted)} \qquad \{P * R\}\ e\ \{v.\, P * Q\} \qquad \text{atomic}(e)}{\left\{\boxed{P}^{\iota} * R\right\}\ e\ \left\{v.\, Q\right\}}$$

Fig. 9. Invariant rules.

rule CountSep lets us join (or conversely, split) two stakes by summing their permissions and their count values, subject to the (implicit) constraint that q, q', and $q + q'$ all lie in the interval $(0, 1]$. The CountAlloc rule lets us create a new counter with some existentially quantified name; the $\Rrightarrow$ connective here is a kind of implication in Iris which lets one modify or create a resource. Finally, CountUpd lets us modify a counter: if we own the global value and a stake, we can update the value and the stake, so long as we preserve the part of the counter value owned by other stakes (represented by k in the rule).

Of course, we need some way to connect these "abstract" resources to the actual state of the program. The mechanism for doing this is an *invariant.* An invariant is an assertion that is dynamically established at some point in the program, and then is guaranteed to hold thereafter. We write $\boxed{P}^{\iota}$ for the assertion which says that the invariant P has been established with the abstract name ι. If we have ownership of resources satisfying P, we can use the rule inv-alloc from Figure 9 to establish P as an invariant; we lose the resources and get back $\boxed{P}^{\iota}$ with some fresh name ι. If we know the invariant $\boxed{P}^{\iota}$ holds and we are trying to prove some Hoare triple about an expression e, we can use inv-open to "open" the invariant. This lets us add P to the pre-condition of the triple we are trying to prove, but we need to re-establish and give up P in the post-condition in order to "close" the invariant. Moreover, to use this rule, e must be atomic: Since e will reduce to a value in a single-step, this ensures there is no intermediate step in which the invariant did not hold. (In the statement of inv-open we have omitted certain side conditions that are used to ensure that the same invariant is not opened multiple times simultaneously.)

For example, we can create the invariant $\boxed{\exists n.\, l \hookrightarrow n * \boxed{\bullet\, n}^{\gamma}}^{\iota}$ to ensure the physical heap location l will always store the value represented by the counter resource. Now a thread that owns a stake $\boxed{\circ\,(q, n')}^{\gamma}$ can read from l or modify it using a compare-and-swap by opening the invariant and updating the global counter resource suitably with CountUpd. This resource and invariant pattern were used to verify a (non-approximate) concurrent counter in the Iris Coq development. As we shall see, we are able to use this same resource to verify the approximate counter with our extensions.

3.3 Probabilistic Rules

We now describe how we extend Iris with probabilistic relational reasoning. Our goal is to be able to prove that there exists a suitable coupling between the indexed valuation of a program and some set $\mathcal{I}$ of indexed valuations. The existence of an appropriate coupling will let us use Theorem 2.1, so that we can bound the expected values of our program by bounding the extrema of $\mathcal{I}$.

To motivate the rules of our extension, let us first give some more background on how this kind of relational reasoning works in the pRHL logic of Barthe et al.. The idea there, following Benton's Relational Hoare Logic [12], is to replace Hoare triples with Hoare *quadruples*, which replace pre and post-conditions with pre and post-relations about *pairs* of programs. Translated to our setting,

$$\frac{0 \le n_1/n_2 \le 1 \qquad (\mathsf{ret\ True} \oplus_{\frac{n_1}{n_2}} \mathsf{ret\ False}) \sim \mathcal{I} : R}{\{\mathsf{Prob}(x \leftarrow \mathcal{I}; F(x))\}\ \mathsf{flip}(n_1, n_2)\ \{v.\, \exists v'.\, \mathsf{Prob}(F(v')) \wedge R(v, v')\}}\ \textsc{Ht-Couple}$$

$$\frac{0 \le n_1/n_2 \le 1}{\{\mathsf{True}\}\ \mathsf{flip}(n_1, n_2)\ \{v.\, (v = \mathsf{True} \vee v = \mathsf{False})\}}\ \textsc{Ht-NonCouple} \qquad \frac{\mathcal{I}' \subseteq \mathcal{I}}{\mathsf{Prob}(\mathcal{I}) \Rightarrow \mathsf{Prob}(\mathcal{I}')}\ \textsc{ProbLe}$$

Fig. 10. Probabilistic rules.

we would have judgments of the form:

$$\{P\}\ e\ -\ \mathcal{I}\ \{x, y.\, Q\}$$

where e is the program we are trying to relate to $\mathcal{I}$, and in the post-relation Q, we would substitute the return value of e in for x and the return value of $\mathcal{I}$ for y. Then, we would adapt the standard Hoare rules to consider the pairs of steps of e and $\mathcal{I}$. Although the work of Barthe et al. shows that this approach can be useful for reasoning about non-concurrent probabilistic programs, there is an issue with applying it in the concurrent setting: what do we do when e forks a new thread? We would then need to relate $\mathcal{I}$ to multiple program expressions, and it's not clear how to adapt ML-FORK to this quadruple style.

Instead, we adapt an idea originally developed by Turon et al. [56] for non-probabilistic concurrent relational reasoning: rather than including the specification program $\mathcal{I}$ as part of the Hoare judgment, we add a new assertion $\mathsf{Prob}(\mathcal{I})$ to the logic. That is, the specification computation becomes just another kind of "resource" that can be transferred between threads. We then add the probabilistic rules shown in Figure 10. The rule HT-COUPLE lets us establish a triple about a $\mathsf{flip}(n_1, n_2)$ command. The precondition requires us to own a monadic computation of the form $x \leftarrow \mathcal{I}\ ;\ F(x)$, and we must exhibit a coupling between a random choice between True and False, (weighted by $\frac{n_1}{n_2}$) and the monadic specification $\mathcal{I}$. The post condition says that we get back the monadic resource, but updated so that it is now of the form $\mathsf{Prob}(F(v'))$ for some v'. In addition, v (the outcome of the $\mathsf{flip}(n_1, n_2)$ command) and this v' are related by R, the postcondition of the coupling we exhibited. This rule gives us a way to relate the execution of a concrete expression e to an execution of $\mathcal{I}$.

Rule HT-NONCOUPLE lets us handle a case where the concrete program executes a probabilistic choice yet we do not want to relate this to a step in the probabilistic specification. In this case, in the post-condition we merely know that the return value was True or False.

Finally, PROBLE lets us replace our $\mathcal{I}$ resource with any $\mathcal{I}'$ such that $\mathcal{I}' \subseteq \mathcal{I}$. We use this to manipulate the $\mathcal{I}$ into a form that matches the precondition required by something like HT-COUPLE. Since $\mathcal{I}' \equiv \mathbb{I}$ implies $\mathcal{I}' \le \mathcal{I}$, it follows that if $\mathcal{I}' \equiv \mathcal{I}$ then $\mathsf{Prob}(\mathcal{I}) \Leftrightarrow \mathsf{Prob}(\mathcal{I}')$.

Because $\mathsf{Prob}(\mathcal{I})$ is just an assertion like any other, we can control access to it between threads by storing it in an invariant. This idea of representing a specification computation as a resource assertion has been used in other separation logics based on Iris [37, 52].

3.4 Soundness

The following soundness theorem for the logic will guarantee that if we prove an appropriate triple involving $\mathrm{Prob}(\mathcal{I})$, the expected value of the concrete program will lie in the range of the extrema of $\mathcal{I}$:

Theorem 3.1. *Let $\mathcal{I} : M_N(M_I(T))$ for some type T, and let $f : Val \to \mathbb{R}$, $g : T \to \mathbb{R}$, and assume that g is bounded on the support of $\mathcal{I}$. Suppose*

$$\{Prob(\mathcal{I})\}\ e\ \{v.\, \exists v'.\, Prob(ret\ v') \land f(v) = g(v')\}$$

holds. Let φ be a well-formed scheduler such that $([e], \sigma)$ terminates in at most n steps under φ. Then $\mathbb{E}_f[resStep^n_\varphi([e], \sigma)]$ exists and

$$\mathbb{E}^{min}_g[\mathcal{I}] \le \mathbb{E}_f[resStep^n_\varphi([e], \sigma)] \le \mathbb{E}^{max}_g[\mathcal{I}]$$

Let us comment on a few aspects of this result. First, it only holds for schedulers under which the program is guaranteed to terminate in some number of steps; this is not that surprising, since the original Iris is a partial correctness logic. Second, it only holds for well-formed schedulers – those that only select threads which can in fact take steps – however, since the normal soundness theorem for Iris implies that e will not get stuck, and neither will any other threads it creates, the well-formedness requirement just means that φ will not try to step a non-existent thread id or a thread that has already terminated in a value.

To prove this soundness theorem and validate the rules we have given, we first change the definition of the Hoare triple in Iris so that if the probabilistic resource is of the form $\mathrm{Prob}(x \leftarrow \mathcal{I}\ ;\ F(x))$ and the expression e takes a step, we must exhibit a coupling between e's transition (interpreted as an indexed valuation) and $\mathcal{I}$. Then in the soundness proof, as e takes successive steps, we combine these couplings together using Bind from Figure 5; if e terminates and the post-condition matches the form stated in Theorem 3.1, then we will have constructed a complete coupling between $\mathrm{resStep}^n_\varphi([e], \sigma)$ and the monadic specification. Moreover, this will be an R-coupling with $R(x, y) \triangleq f(x) = g(y)$. Hence, we can apply Theorem 2.1 to conclude the claim about the expected values.

There is one caveat: in Ht-NonCouple, the program takes a probabilistic step but the premise does not require exhibiting a coupling. To support this, we have to construct a kind of trivial "dummy" coupling that preserves the expected values. We describe this further in Appendix A.

4 EXAMPLE 1: APPROXIMATE COUNTERS

In this section we prove triples that relate the approximate counter algorithm from Figure 1c to the monadic computation approxN from Figure 3.

4.1 Triples and Example Client

The Hoare triples we have proved about this data structure are given in Figure 11. The specification uses a predicate $\mathrm{ACounter}_{\gamma_l, \gamma_p, \gamma_c}(l, q, n)$, which can be treated by a user as an abstract predicate representing the permission to perform n increments to the counter at l. The parameter q is a fractional permission that we use to track how many threads can access the counter. (Ignore the names γ_l, γ_p, and γ_c – we will describe how they are used when we give the definition of ACounter later). The triple ACounterNew says that we can create a new counter by allocating a reference cell containing 0. It takes the monadic specification $\mathrm{Prob}(\mathrm{approxN}\ n\ 0)$ as a precondition, and returns the full ACounter permission for n increments. The rule ACounterSep lets us split or join this ACounter permission into pieces. If we have permission to perform at least one increment, we can use

ACounterNew
$$\{\mathsf{Prob}(\mathsf{approxN}\ n\ 0)\}\ \mathsf{ref}\ 0\ \{l.\ \exists \gamma_{\mathrm{l}}, \gamma_{\mathrm{p}}, \gamma_{\mathrm{c}}, \mathsf{ACounter}_{\gamma_{\mathrm{l}},\gamma_{\mathrm{p}},\gamma_{\mathrm{c}}}(l, 1, n)\}$$

ACounterSep
$$\mathsf{ACounter}_{\gamma_{\mathrm{l}},\gamma_{\mathrm{p}},\gamma_{\mathrm{c}}}(l, q+q', n+n') \Leftrightarrow \mathsf{ACounter}_{\gamma_{\mathrm{l}},\gamma_{\mathrm{p}},\gamma_{\mathrm{c}}}(l, q, n) * \mathsf{ACounter}_{\gamma_{\mathrm{l}},\gamma_{\mathrm{p}},\gamma_{\mathrm{c}}}(l, q', n')$$

ACounterIncr
$$\{\mathsf{ACounter}_{\gamma_{\mathrm{l}},\gamma_{\mathrm{p}},\gamma_{\mathrm{c}}}(l, q, n+1)\}\ \mathsf{incr}\ l\ \{\mathsf{ACounter}_{\gamma_{\mathrm{l}},\gamma_{\mathrm{p}},\gamma_{\mathrm{c}}}(l, q, n)\}$$

ACounterRead
$$\{\mathsf{ACounter}_{\gamma_{\mathrm{l}},\gamma_{\mathrm{p}},\gamma_{\mathrm{c}}}(l, 1, 0)\}\ \mathsf{read}\ l\ \{v.\ \exists v'.\ \mathsf{Prob}(\mathsf{ret}\ v') \land v = v'\}$$

Fig. 11. Specification for approximate counters.

$$\mathsf{countTrue}\ c\ lb \triangleq \mathsf{foldLeft}\ (\lambda\ \_\ b.\ \mathsf{if}\ b\ \mathsf{then}\ (\mathsf{incr}\ c)\ \mathsf{else}\ ())\ lb\ ()$$

$$\{\mathsf{Prob}(\mathsf{approxN}\ (|lb_1|_{\mathrm{t}} + |lb_2|_{\mathrm{t}})\ 0)\}$$
$$\mathsf{let}\ c = \mathsf{ref}\ 0\ \mathsf{in}$$
$$\{\mathsf{ACounter}_{\gamma_{\mathrm{l}},\gamma_{\mathrm{p}},\gamma_{\mathrm{c}}}(c, 1, |lb_1|_{\mathrm{t}} + |lb_2|_{\mathrm{t}})\}$$
$$\begin{array}{c||c} \{\mathsf{ACounter}_{\gamma_{\mathrm{l}},\gamma_{\mathrm{p}},\gamma_{\mathrm{c}}}(c, 1/2, |lb_1|_{\mathrm{t}})\} & \{\mathsf{ACounter}_{\gamma_{\mathrm{l}},\gamma_{\mathrm{p}},\gamma_{\mathrm{c}}}(c, 1/2, |lb_2|_{\mathrm{t}})\} \\ \mathsf{countTrue}\ c\ lb_1 & \mathsf{countTrue}\ c\ lb_2 \\ \{\mathsf{ACounter}_{\gamma_{\mathrm{l}},\gamma_{\mathrm{p}},\gamma_{\mathrm{c}}}(c, 1/2, 0)\} & \{\mathsf{ACounter}_{\gamma_{\mathrm{l}},\gamma_{\mathrm{p}},\gamma_{\mathrm{c}}}(c, 1/2, 0)\} \end{array}$$
$$\{\mathsf{ACounter}_{\gamma_{\mathrm{l}},\gamma_{\mathrm{p}},\gamma_{\mathrm{c}}}(c, 1, 0)\}$$
$$\mathsf{read}\ c$$
$$\{v.\ \exists v'.\ \mathsf{Prob}(\mathsf{ret}\ v') \land v = v'\}$$

Fig. 12. Example client using approximate counters.

$$\mathsf{LocInv}_{\gamma_{\mathrm{l}}}(l) \triangleq \exists n.\ l \hookrightarrow n * \boxed{\bullet\, n}^{\gamma_{\mathrm{l}}}$$

$$\mathsf{ProbInv}_{\gamma_{\mathrm{p}},\gamma_{\mathrm{c}}} \triangleq \exists n_1, n_2.\ \boxed{\bullet\, n_1}^{\gamma_{\mathrm{p}}} * \boxed{\bullet\, n_2}^{\gamma_{\mathrm{c}}} * (\mathsf{Prob}(\mathsf{approxN}\ n_1\ n_2) \lor \boxed{\circ\,(1, n_1)}^{\gamma_{\mathrm{p}}})$$

$$\mathsf{ACounter}_{\gamma_{\mathrm{l}},\gamma_{\mathrm{p}},\gamma_{\mathrm{c}}}(l, q, n) \triangleq \exists \iota_1, \iota_2, n'.\ \boxed{\mathsf{LocInv}_{\gamma_{\mathrm{l}}}(l)}^{\iota_1} * \boxed{\mathsf{ProbInv}_{\gamma_{\mathrm{p}},\gamma_{\mathrm{c}}}}^{\iota_2} * \boxed{\circ\,(q, n')}^{\gamma_{\mathrm{l}}} * \boxed{\circ\,(q, n)}^{\gamma_{\mathrm{p}}} * \boxed{\circ\,(q, n')}^{\gamma_{\mathrm{c}}}$$

Fig. 13. Invariants and definitions for proof.

ACounterIncr, which gives us back ACounter with permission to do one fewer increment. Finally, if we have ACounter with the full fractional permission 1, and there are 0 pending increments, we can use ACounterRead. In the post condition we get back $\mathsf{Prob}(\mathsf{ret}\ v)$, where v is the value that the call to read returns.

At first this specification seems weak, but this is exactly what we need for Theorem 3.1. To see how we can use these triples to reason about a client program that uses the approximate counter, consider the example client in Figure 12. We start with a helper function countTrue, which takes an approximate counter c and a list of booleans lb, and counts the number of times True occurs in lb using the counter. The client begins by creating a new counter c. It then runs two threads in parallel that run countTrue on two lists lb_1 and lb_2, using the shared counter c – we denote this parallel composition using $\|$. The parent blocks until both threads finish and then reads from the counter[7].

Refer to this client code as e. If we write $|lb|_{\mathsf{t}}$ for the logical function giving the number of times True occurs in lb, then we would like to show that in expectation, e returns $|lb_1|_{\mathsf{t}} + |lb_2|_{\mathsf{t}}$. The derivation in Figure 11 shows that the triple

$$\{\mathsf{Prob}(\mathsf{approxN}\ (|lb_1|_{\mathsf{t}} + |lb_2|_{\mathsf{t}})\ 0)\}\ e$$
$$\{v.\ \exists v'.\ \mathsf{Prob}(\mathsf{ret}\ v') \wedge v = v'\}$$

holds. Moreover, it is not hard to show that for each k, there is an upper bound on the value returned by approxN k, so by Theorem 3.1 we have:

$$\mathbb{E}^{\mathsf{min}}_{\mathsf{id}}[\mathsf{approxN}\ (|lb_1|_{\mathsf{t}} + |lb_2|_{\mathsf{t}})] \leq \mathbb{E}_{\mathsf{id}}[\mathsf{resStep}^{n}_{\varphi}([e], \sigma)]$$

(and similarly for $\mathbb{E}^{\mathsf{max}}$) for suitable φ and n. And, we have shown that $\mathbb{E}^{\mathsf{min}}_{\mathsf{id}}[\mathsf{approxN}\ (|lb_1|_{\mathsf{t}} + |lb_2|_{\mathsf{t}})] = |lb_1|_{\mathsf{t}} + |lb_2|_{\mathsf{t}}$ in §2.4, so we are done.

4.2 Proofs of Triples

The definition of ACounter and the invariants used in the proof are given in Figure 13. The proof uses three counter resources to track: (1) the number of increments left to perform in the monadic specification, (2) the accumulated count in the monadic specification, and (3) the actual count currently stored in the concrete program. We use two invariants to connect the counter resources to these intended interpretations. First, we have $\mathsf{LocInv}_{\gamma_{\mathsf{l}}}(l)$ which says that the counter resource named γ_{l} stores some value n and the physical location l points to that same value n. Then, assertion $\mathsf{ProbInv}_{\gamma_{\mathsf{p}},\gamma_{\mathsf{c}}}$ says that there are two counter resources containing some n_1 and n_2, and the invariant either contains (a) the monadic specification resource $\mathsf{Prob}(\mathsf{approxN}\ n_1\ n_2)$ (*i.e.,* there are n_1 further increments to perform, and the monadic counter has accumulated a value of n_2) *or* (b) it contains the complete stake for one of the counter resources. Then ACounter says that these two invariants have been set up with some names, and we own a stake in the γ_{p} permission corresponding to the number of increments this permission allows. Further, for some n' there is a stake in the γ_{l} and γ_{c} counters both equal to n', which represents the total amount that this permission has been used to add to the counter.

We will only describe the proofs of ACounterIncr and ACounterRead, since ACounterNew is straight-forward.

Proof of ACounterIncr. Eliminating the existentials in the definition of ACounter, we get that the appropriate invariants have been set up and there is some n'-stake in γ_{l} and γ_{c}, along with the $n+1$ stake in γ_{p}. The first step of incr l reads the value of l; to perform this read the thread needs to own $l \hookrightarrow v$ for some v. To get this resource, it opens the $\mathsf{LocInv}_{\gamma_{\mathsf{l}}}(l)$ invariant; after completing

[7] Of course, here the threads may as well maintain their own exact counters and combine them at the end. But in a real application such as [55], there are tens of millions of counters and hundreds of threads, so having each thread maintain its own set of counters would be expensive.

the read, the $l \hookrightarrow v$ resource is returned to close the invariant. The code then takes the minimum of the value read and MAX, and binds this value to k.

It then performs flip$(1, k+1)$. We want to use Ht-Couple to couple this flip with the monadic code. To do so, we first open the invariant $\mathsf{ProbInv}_{\gamma_p,\gamma_c}$. We know this will contain $\boxed{\bullet\, n_1'}^{\gamma_p}$ and $\boxed{\bullet\, n_2'}^{\gamma_c}$ for some n_1' and n_2', and either $\mathsf{Prob}(\mathsf{approxN}\ n_1'\ n_2')$ or a full stake $\boxed{\circ\,(1, n_1')}^{\gamma_p}$. However, the latter is impossible because the $\mathsf{ACounter}_{\gamma_l,\gamma_p,\gamma_c}(l, q, n+1)$ resource entails ownership of $\boxed{\circ\,(q, n+1)}^{\gamma_p}$, but $q+1 > 1$, contradicting CountPerm. So, we obtain $\mathsf{Prob}(\mathsf{approxN}\ n_1'\ n_2')$. Now, by CountGeq we know that $n_1' \geq n+1$, hence we can unfold $\mathsf{approxN}\ n_1'\ n_2'$ to obtain $\mathsf{Prob}(k \leftarrow \mathsf{approxIncr}\,;\ \mathsf{approxN}\ (n_1' - 1)\ n_2')$.

We can now use Ht-Couple so long as we can exhibit a coupling between the concrete program's coin flip and approxIncr. First, since $0 \leq k \leq \mathsf{MAX}$, we can show that:

$$\begin{aligned}
&(\mathsf{ret}\ k+1) \oplus_{\frac{1}{k+1}} (\mathsf{ret}\ 0) \\
&\quad \subseteq (x \leftarrow \mathsf{ret}\ 0 \cup \cdots \cup \mathsf{ret}\ \mathsf{MAX}\,;\ (\mathsf{ret}\ x+1 \oplus_{\frac{1}{x+1}} \mathsf{ret}\ 0)) \\
&\quad \equiv \mathsf{approxIncr}
\end{aligned}$$

hence by Equiv, it suffices to exhibit a coupling between $(\mathsf{ret}\ \mathsf{True} \oplus_{\frac{1}{k+1}} \mathsf{ret}\ \mathsf{False})$ and $(\mathsf{ret}\ k+1 \oplus_{\frac{1}{k+1}} \mathsf{ret}\ 0)$. Take $R(x, y)$ to be $(x = \mathsf{True} \wedge y = k+1) \vee (x = \mathsf{False} \wedge y = 0)$, then we can use P-Choice and Ret to prove the existence of an R-coupling.

Applying Ht-Couple with this coupling, we then have $\mathsf{Prob}(\mathsf{approxN}\ (n_1' - 1)\ (n_2' + v'))$ where v' and the return value v of the flip$(1, k)$ are related by R. We use CountUpd to update the thread's stake in γ_p resource to n, and the global value to $n_1' - 1$ (to record that a simulated increment has performed), similarly, we update the thread's stake in the γ_c counter to $n' + v'$ and the global value to $n_2' + v'$ (to record the new total) and then close the $\mathsf{ProbInv}_{\gamma_p,\gamma_c}$ invariant.

The code then cases on the value v returned by the flip. If it is false, then v' is 0, the code returns, and the post condition holds. If v is true, then $v' = k+1$, the amount that the code adds using a fetch-and-add. We therefore open the $\mathsf{LocInv}_{\gamma_l}(l)$ invariant again to get access to l, perform the increment and update the γ_l counter and stake using CountUpd to record the fact that we are adding $k+1$.

Proof of ACounterRead. The precondition $\mathsf{ACounter}_{\gamma_l,\gamma_p,\gamma_c}(1, 0, n_2)$ represents the full stake in each counter, and the 0 second argument means there are no pending increments to perform. Thus, when we open the $\mathsf{LocInv}_{\gamma_l}(l)$ and $\mathsf{ProbInv}_{\gamma_p,\gamma_c}$ invariants we know that $l \hookrightarrow n_2$ and we have $\mathsf{Prob}(\mathsf{ret}\ n_2)$. So, we can read from l, knowing the returned value will be n_2. After reading, we must close the invariant: this time we will keep the $\mathsf{Prob}(\mathsf{ret}\ n_2)$ resource so that we can put it in the post condition, instead we give up $\boxed{\circ\,(1, 0)}^{\gamma_p}$ to satisfy the disjunction in $\mathsf{ProbInv}_{\gamma_p,\gamma_c}$.

4.3 Variations

In our mechanized proofs, we have verified two additional variations on this approximate counter example. For the first variation, we consider a version of incr which directly uses the current value it reads from the counter, rather than taking the minimum of this value and MAX.

For the second variation, we address a limitation of the specification we have described so far. Notice that to use rules triples in Figure 11 and obtain a suitable triple to use with Theorem 3.1, the total number of calls to incr must be a deterministic function of the program : we have to pick some n when we initialize the counter using ACounterNew. In the case of our example client, we chose n to be the number of times that true occurred in the two lists. But what if the number of calls to increment is itself probabilistic or non-deterministic? In this case we still would like to know that the expected value returned by the approximate counter is equal to the expected number of times the counter was incremented. However, if the number of times the counter is incremented is completely arbitrary, this expected value may not exist! To guarantee that the expected value will exist, our specification imposes an upper bound on the total number of increments that can be performed, and then allows us to establish a coupling with the following monadic computation:

$$\begin{aligned}&\mathsf{approxN}'\ 0\ t\ l \triangleq \mathsf{ret}\ (t, l)\\&\mathsf{approxN}'\ (n+1)\ t\ l \triangleq (\mathsf{ret}\ (t, l)) \cup (k \leftarrow \mathsf{approxIncr}\,;\ \mathsf{approxN}'\ n\ (t+1)\ (l+k))\end{aligned}$$

The first argument of $\mathsf{approxN}'$ gives an upper bound on the remaining number of increments that can be performed, the second argument t tracks the total number of increments that have been done so far, and l again tracks the current value in the counter. In contrast to the original approxN, before performing each increment, there is a non-deterministic choice to simply return (t, l). Let f be the function $\lambda(x, y).\ x - y$. We prove that

$$\mathbb{E}_f^{\mathsf{min}}[\mathsf{approxN}'\ n\ 0\ 0] = \mathbb{E}_f^{\mathsf{max}}[\mathsf{approxN}'\ n\ 0\ 0] = 0$$

i.e., the expected value of the difference between the total number of increments and the value in the counter is 0. We have proved more flexible versions of the rules in Figure 11 that use this $\mathsf{approxN}'$ instead.

5 EXAMPLE 2: CONCURRENT SKIP LIST

For our next example, we verify properties of a probabilistic concurrent skip list. The code and proofs for this example are more complex, so for space reasons we will give a high-level description of the algorithm and the triples we have proved about it.

5.1 Implementation

A skip list [48] is a data structure that can be used to implement a dynamic set interface for ordered data. The implementation we consider will only allow integer keys to be stored in the set. The skip list consists of several sorted linked lists, where the nodes in each list contain a key. We visualize each list as running horizontally from left to right, with the different lists stacked vertically above one another (see Figure 14). For simplicity we only allow 2 lists in our implementation – this still exposes most of the main concurrency issues involved. The set of keys contained in the top list is a subset of the keys contained in the bottom list, and the node containing a key k in the top list includes a pointer to the corresponding node for k in the list below it. At the beginning and ends of each list, there are sentinel nodes containing the minimum and maximum representable integer (which we write as $-\infty$ and $+\infty$ in Figure 14).

Non-concurrent implementation. We first describe how operations on this data structure are implemented in the sequential case. To check whether a key k is contained in the set, we first search for the key in the top list starting at the left sentinel. If we find a node containing it, we return true. If not, we stop at the largest key $j < k$ in the list, and then follow the pointer in j's node to the copy of j in the bottom list. We then resume searching for k starting at node j in the bottom list. If k is found in the bottom list we return true, otherwise the key is not in the set so we return false. To insert k, we first find the nodes N_t and N_b with the largest keys less than $\leq k$ in the top and bottom list, respectively. If we find that k is already in either list, we stop and return. Otherwise we execute $\mathsf{flip}(1/2)$. If it returns true, we insert new nodes for key k into both the top and bottom lists, after N_t and N_b. Otherwise, if it returns false, we only insert a node in the bottom list after N_b. We call N_t and N_b the "predecessor nodes", because they become the predecessors of k if it is inserted into each list.

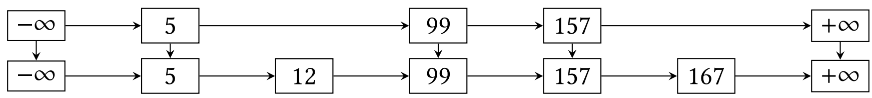

Fig. 14. Diagram for 2-Level Concurrent Skip List

If we insert n keys into the set, then in expectation $n/2$ of them will appear in the top list. Then when searching for a key, we will be able to more quickly descend down the top list, and either find they key there, or if not, only have to examine a few additional nodes in the bottom list. Of course, it is possible (though unlikely) that none or all of the nodes are inserted into the top list, in which case we are effectively searching in a regular sorted linked list. Later on, we will show how to use our program logic to derive a bound on the expected number of comparisons needed to find a key in the list. We will not handle deletion in our implementation, because if an adversarial client can observe the state of the list, it can repeatedly delete and re-insert any key that happens to end up in the top list, forcing the top list to be empty.

Adding concurrency. There are several ways to add support for concurrent operations to a skip list. We will consider a simplified implementation inspired by that of Herlihy et al. [28]. We add a lock to each node in the lists. Checking for whether a key is in the set is the same as in the non-concurrent case, and no locks need to be acquired.

To insert a key k, we again search for the predecessor nodes N_t and N_b. When we identify one of these nodes, we acquire its lock and then check that the node after it has not changed in the time between when we examined its successor and when the lock was acquired. If it has, that means another thread may have inserted a new node with a larger key less than k, so we release the lock and search for the predecessor again. Otherwise, so long as we hold the locks, we are guaranteed that N_t and N_b will remain the proper predecessors for key k. Having acquired both locks, we proceed as in the sequential case by generating a random bit, and on the basis of that bit we insert new nodes for k into either both lists or just the bottom list. We then release the locks and return.

5.2 Monadic Model

We follow the same pattern as in our verification of the approximate counter example: we first define a monadic model of the data structure, bound appropriate expected values of the monadic computation, and then describe triples that can be used to prove the existence of a coupling between programs using the skip list and the monadic model.

Our monadic model is the following:

$$\begin{aligned}
\mathsf{skiplist}\ \mathsf{nil}\ tl\ bl \quad &\triangleq \quad \mathsf{ret}\ (\mathsf{sort}(tl), \mathsf{sort}(bl)) \\
\mathsf{skiplist}\ (k :: l)\ tl\ bl \quad &\triangleq \quad k' \leftarrow \bigcup_{i \in k::l} \mathsf{ret}\ i\,; \\
&\qquad\ tl' \leftarrow (\mathsf{ret}\ tl) \oplus_{1/2} (\mathsf{ret}\ k' :: tl)\,; \\
&\qquad\ \mathsf{skiplist}\ (\mathsf{remove}\ k'\ (k :: l))\ tl'\ (k' :: bl)
\end{aligned}$$

The computation skiplist $l\ tl\ bl$ simulates adding keys from the list l to a skip list, where the arguments tl and bl represent the keys in the top and bottom lists of the skip list, respectively. If the first argument l is empty, it sorts tl and bl and returns the result. If l is non-empty, it first non-deterministically selects a key k' from l. Then, with probability 1/2 it adds this key to tl. It then removes any copies of k' from l, and recurses to process the remaining elements with the updated top and bottom lists. (There is no point in keeping the arguments tl and bl sorted throughout the recursive calls in this monadic formulation.)

There are many quantitative properties of the skip list that one might want to analyze (*e.g.*, the total number of nodes in both lists, the probability that a large fraction of nodes lie only in the bottom list, *etc.*). As we alluded to above, we will bound the expected number of inequality comparisons required to test for membership of a key in the skiplist. We define a function $\mathsf{skipcost}(tl, bl, k)$ which gives the number of comparisons needed to check if k is in the skip list when the elements in the top and bottom lists are tl and bl, respectively:

$$\begin{aligned}
\mathsf{topcost}(tl, k) &= 1 + |\{i \in tl \mid \mathsf{INTMIN} < i < k\}| \\
\mathsf{maxbelow}(tl, k) &= \max(\{i \in tl \mid i < k\} \cup \{\mathsf{INTMIN}\}) \\
\mathsf{botcost}(tl, bl, k) &= 1 + |\{i \in bl \mid \mathsf{maxbelow}(tl, k) < i < k\}| \\
\mathsf{skipcost}(tl, bl, k) &= \begin{cases} \mathsf{topcost}(tl, k) & \text{if } k \in tl \\ \mathsf{topcost}(tl, k) + \mathsf{botcost}(tl, bl, k) & \text{if } k \notin tl \end{cases}
\end{aligned}$$

If the key k is in the top list, then the number of comparisons is 1 plus the number of elements in the list less than k ($\mathsf{topcost}(tl, k)$). If k is not in the top list, then we must first still perform the same number of comparisons while searching through the top list. Then we search in the bottom list starting from the largest key less than k that was in tl ($\mathsf{maxbelow}(tl, k)$). The total number of comparisons in the second list is the number of keys between $\mathsf{maxbelow}(tl, k)$ and k ($\mathsf{botcost}(tl, bl, k)$).

We then bound $\mathbb{E}^{\max}_{\mathsf{skipcost}(-,-,k)}[\mathsf{skiplist}\ l\ \mathsf{nil}\ \mathsf{nil}]$, to obtain an upper bound on the expected value of searching for a key k. Assuming l has no duplicates, the key k and all keys in l lie between INTMIN and INTMAX, and there are n keys less than k in l, we show that:

$$\mathbb{E}^{\max}_{\mathsf{skipcost}(-,-,k)}[\mathsf{skiplist}\ l\ \mathsf{nil}\ \mathsf{nil}] \leq 1 + \frac{n}{2} + 2\left(1 - \frac{1}{2^{n+1}}\right)$$

This means that on average we have to do about half the number of comparisons that would be required to search for the key in a regular sorted linked list.

5.3 Triples

Figure 15 shows the triples we have proved about the skip list. The specification defines an assertion $\mathsf{SkipPerm}_\Gamma(q, v, S, S_t, S_b)$, which represents permission to access a skip list whose top left sentinel is v. The argument Γ is just a set of resource names (like the γ's in the counter example), q is a fractional permission, S is the finite set of keys which may be added to the list, and S_t and S_b are a subset of the keys currently in the top and bottom lists. Additional keys from S may be in either S_t or S_b, but the owner of this permission assertion knows that they contain *at least* these sets.

The expression $\mathsf{newSkipList}$ creates a new skip list. The precondition for the triple in SkipNew requires us to own the monadic computation[8] $\mathsf{Prob}(\mathsf{skiplist}\ S\ \mathsf{nil}\ \mathsf{nil})$. The post condition gives the full permission ($q = 1$) to access the skip list, with empty top and bottom lists. To use this rule, all the keys in S must be between INTMIN and INTMAX. Notice here that the set of keys S which will be added to the skiplist must be deterministic, so that it can be decided in this precondition (much like our original specification for the approximate counters required the total number of increments to be deterministic). This restriction is important: if the keys to be added are non-deterministically selected, and a client can observe the state of the skip list, it can insert a special sequence of keys in such a way so as to force a large number of comparisons to find a particular target key.

[8] Here, S is a set, whereas the arguments to skiplist are lists. However, it is easy to show that if l', tl', and bl' are permutations of the lists l, tl, and bl, respectively, then $\mathsf{skiplist}\ l\ tl\ bl \equiv \mathsf{skiplist}\ l'\ tl'\ bl'$, so it makes no difference if we treat the first argument instead as an unordered set.

SkipNew
$$\frac{\forall k \in S.\, \mathsf{INTMIN} < k < \mathsf{INTMAX}}{\{\mathsf{Prob}(\mathsf{skiplist}\ S\ \mathsf{nil}\ \mathsf{nil})\}\ \mathsf{newSkipList}\ \{v.\, \exists \Gamma.\, \mathsf{SkipPerm}_\Gamma(1, v, S, \emptyset, \emptyset)\}}$$

SkipAdd
$$\frac{k \in S}{\{\mathsf{SkipPerm}_\Gamma(q, v, S, S_t, S_b)\}\ \mathsf{addSkipList}\ v\ k\ \{\_.\ \exists S_t'.\, \mathsf{SkipPerm}_\Gamma(q, v, S, S_t', S_b \cup \{k\})\}}$$

SkipMem
$$\frac{\mathsf{INTMIN} < k < \mathsf{INTMAX}}{\begin{array}{l}\{\mathsf{SkipPerm}_\Gamma(1, v, S, S_t, S_b)\}\ \mathsf{memSkipList}\ v\ k \\ \left\{(b, z).\begin{array}{l}\mathsf{SkipPerm}_\Gamma(1, v, S, S_t, S_b) * (b = \mathsf{True} \Rightarrow k \in S_b) \\ * (b = \mathsf{False} \Rightarrow k \notin S_t \cup S_b) * (z = \mathsf{skipcost}(S_t, S_b, k))\end{array}\right\}\end{array}}$$

SkipSep
$$\mathsf{SkipPerm}_\Gamma(q + q', v, S, S_t \cup S_t', S_b \cup S_b') \Leftrightarrow \mathsf{SkipPerm}_\Gamma(q, v, S, S_t, S_b) * \mathsf{SkipPerm}_\Gamma(q', v, S, S_t', S_b')$$

SkipShift
$$\mathsf{SkipPerm}_\Gamma(1, v, S, S_t, S) \Rrightarrow \mathsf{Prob}(\mathsf{ret}\ (\mathsf{sort}(S_t), \mathsf{sort}(S)))$$

Fig. 15. Specification for skip list.

We use addSkipList to insert a key k into the skip list. The post condition in SkipAdd indicates that we now know that the added key k is in the bottom list. On the other hand, the client does not know whether the key was added to the top list or not, so in the permission for the post condition, the contents of the top list are given by some existentially quantified S_t'.

The function memSkipList checks whether a key is in the skip list. It returns a pair (b, z), where b is a boolean indicating whether the key was in the set or not, and z is the number of key comparisons performed. The triple SkipMem says that if we have the full permission for the skip list, then the boolean b indeed reflects whether the key is in the set or not, and z is in fact equal to the cost function $\mathsf{skipcost}(S_t, S_b, k)$ we defined above. (One can also prove triples for when we have less than the full permission of the list (*i.e.*, $q < 1$), but we have not done so.)

The rule SkipSep lets us split and join together the SkipPerm permission so that separate threads can use the skip list. Finally, SkipShift lets us do a view shift to convert a full SkipPerm permission in which we have added all the keys in S to the skip list back into a Prob permission in which the monadic computation is finished. This lets us prove triples of the form required by Theorem 3.1.

In our mechanized proofs we have used this specification to verify a simple client in which two threads concurrently add lists of integers to a skip list set, and then after they both finish, one looks up a key using memSkipList and returns the number of comparisons performed.

6 RELATED WORK

As we have described, the key to our approach has been to synthesize ideas from several lines of related work. We now mention further related work.

Probabilistic logics. McIver et al. [39] present a probabilistic version of rely-guarantee logic [31]. They use their logic to verify a "faulty" concurrent Sieve of Eratosthenes, in which threads remove numbers from a list to identify primes, with thread i removing multiples of $(i + 1)$ – however, each thread only probabilistically removes the elements it is supposed to, and the goal is to give a lower bound on the probability that the resulting list only contains primes. Like the original rely-guarantee logic, this logic does not permit local reasoning: one must check stability against rely-guarantee conditions that refer to the global state of the program. More recent concurrency logics have combined rely-guarantee reasoning with the local reasoning features of concurrent separation logic [21, 57]. Iris, and our extensions, incorporate these ideas, which is what enables us to give specifications like those in Figure 15 – clients can use the skip list without having to reason about interference involving the underlying state of the list. Since many of the interesting uses of randomization in the concurrent setting are in implementations of *data structures*, it is important to be able to provide these more abstract specifications.

Recently, Batz et al. [10] have developed a version of (non-concurrent) separation logic for reasoning about sequential probabilistic programs with dynamic memory allocation. They verify an example of a program which probabilistically appends nodes to a list (so that the length of the list is geometrically distributed), and a tree deletion procedure which only probabilistically deletes nodes. Instead of using relational reasoning, assertions in their logic denote probabilities/expected values, and rules are given for computing and bounding these probabilities.

Several program logics for probabilistic reasoning (*e.g.*, [42]) are designed to reason about languages that have primitives for both probabilistic choice and (demonic) non-deterministic choice. However, in that work, non-determinism was not used for modelling concurrency, but rather a program which might be "underspecified" and have multiple possible implementations of a component, of which one is selected non-deterministically, as in Dijkstra's [18] work on GCL.

Barthe et al. [4] were the first to connect the idea of coupling to the kind of probabilistic relational reasoning done in pRHL, an earlier logic by Barthe et al. [8]. Since then, different results from the theory of coupling and variants of couplings have been used to extend pRHL [5, 6, 9, 29].

Iris uses a step-indexed semantic model to support impredicative features of the logic, but we do not make special use of step-indexing in our extensions. However, Aguirre et al. [1] have shown that a kind of step-indexed model can be used to reason about more general kinds of couplings (so-called "shift couplings"). Previously, Bizjak and Birkedal [13] developed a step-indexed logical relation for a higher-order language with random choice.

Denotational semantics. A number of denotational models combining probabilistic and non-deterministic choice have been developed [27, 30, 40, 54, 59, 60]. Our soundness theorem considers a scheduler which *deterministically* selects which thread to run next. Varacca and Winskel [60] showed that their monadic encoding, which we have used in our work, gives an adequate semantic model for an imperative language with this kind of deterministic scheduler. An alternative is to permit the scheduler to also make random choices when selecting which thread to run. Varacca and Winskel show that in this case, an alternative monad developed by Mislove [40] and Tix et al. [54] gives an adequate model. It would be interesting to use this latter monad in our program logic to reason about behavior under probabilistic schedulers.

7 CONCLUSION

We have developed a concurrent program logic that can be used for probabilistic relational reasoning, and have used it to verify two realistic examples of randomized concurrent algorithms. Moreover, we have mechanized all the results described here in Coq by modifying the prior Coq formalization of Iris. The development is included in our supplementary material.

REFERENCES

[1] Alejandro Aguirre, Gilles Barthe, Lars Birkedal, Ales Bizjak, Marco Gaboardi, and Deepak Garg. 2018. Relational Reasoning for Markov Chains in a Probabilistic Guarded Lambda Calculus. In *ESOP*. 214–241.

[2] Andrew W. Appel. 2014. *Program Logics - for Certified Compilers.* Cambridge University Press.

[3] Philippe Audebaud and Christine Paulin-Mohring. 2009. Proofs of randomized algorithms in Coq. *Sci. Comput. Program.* 74, 8 (2009), 568–589.

[4] Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, Léo Stefanesco, and Pierre-Yves Strub. 2015. Relational Reasoning via Probabilistic Coupling. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings.* 387–401.

[5] Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2017. Proving uniformity and independence by self-composition and coupling. In *LPAR*.

[6] Gilles Barthe, Thomas Espitau, Justin Hsu, Tetsuya Sato, and Pierre-Yves Strub. 2017. *-Liftings for Differential Privacy. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*. 102:1–102:12.

[7] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. A Program Logic for Union Bounds. In *ICALP*. 107:1–107:15.

[8] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2012. Probabilistic Relational Hoare Logics for Computer-Aided Security Proofs. In *Mathematics of Program Construction - 11th International Conference, MPC 2012, Madrid, Spain, June 25-27, 2012. Proceedings.* 1–6.

[9] Gilles Barthe, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2017. Coupling proofs are probabilistic product programs. In *POPL*. 161–174.

[10] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. 2018. Quantitative Separation Logic. *CoRR* abs/1802.10467 (2018). arXiv:1802.10467 http://arxiv.org/abs/1802.10467

[11] Jon Beck. 1969. Distributive laws. In *Seminar on Triples and Categorical Homology Theory*, B. Eckmann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 119–140.

[12] Nick Benton. 2004. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. In *POPL*.

[13] Ales Bizjak and Lars Birkedal. 2015. Step-Indexed Logical Relations for Probability. In *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings.* 279–294.

[14] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Tappan Morris, and Nickolai Zeldovich. 2010. An Analysis of Linux Scalability to Many Cores. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings.* 1–16.

[15] John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings.* 55–72.

[16] Adam Chlipala. 2013. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant.* MIT Press. http://mitpress.mit.edu/books/certified-programming-dependent-types

[17] Dave Dice, Yossi Lev, and Mark Moir. 2013. Scalable statistics counters. In *SPAA*. 43–52.

[18] Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975), 453–457.

[19] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: Compositional reasoning for concurrent programs. In *POPL*.

[20] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. 2010. Concurrent abstract predicates. In *ECOOP*. 504–528.

[21] Xinyu Feng. 2009. Local rely-guarantee reasoning. In *POPL*. 315–327.

[22] Philippe Flajolet. 1985. Approximate Counting: A Detailed Analysis. *BIT* 25, 1 (1985), 113–134.

[23] Keir Fraser. 2004. *Practical lock-freedom.* Ph.D. Dissertation. University of Cambridge.

[24] Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. 2010. Reasoning about optimistic concurrency using a program logic for history. In *CONCUR*. 388–402.

[25] Jeremy Gibbons and Ralf Hinze. 2011. Just do it: simple monadic equational reasoning. In *ICFP*. 2–14.

[26] Jean Goubault-Larrecq. 2007. Continuous Previsions. In *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings.* 542–557.

[27] Jean Goubault-Larrecq. 2015. Full abstraction for non-deterministic and probabilistic extensions of PCF I: The angelic cases. *J. Log. Algebr. Meth. Program.* 84, 1 (2015), 155–184.

[28] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2006. A Provably Correct Scalable Concurrent Skip List (Brief Announcement). In *OPODIS*.

[29] J. Hsu. 2017. Probabilistic Couplings for Probabilistic Reasoning. *ArXiv e-prints* (Oct. 2017). arXiv:cs.LO/1710.09951

[30] Claire Jones. 1990. *Probabilistic non-determinism.* Ph.D. Dissertation. University of Edinburgh, UK. http://hdl.handle.net/1842/413
[31] C. B. Jones. 1983. Tentative steps toward a development method for interfering programs. *TOPLAS* 5, 4 (1983), 596–619.
[32] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-Order Ghost State. In *ICFP*.
[33] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. 637–650.
[34] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2016. Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs. In *ESOP*. 364–389.
[35] Dexter Kozen. 1981. Semantics of Probabilistic Programs. *J. Comput. Syst. Sci.* 22, 3 (1981), 328–350.
[36] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP*. 696–723.
[37] Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. 2017. A Relational Model of Types-and-Effects in Higher-Order Concurrent Separation Logic. In *POPL*. 218–231.
[38] T. Lindvall. 2002. *Lectures on the Coupling Method.* Dover Publications, Incorporated.
[39] Annabelle McIver, Tahiry M. Rabehaja, and Georg Struth. 2016. Probabilistic rely-guarantee calculus. *Theor. Comput. Sci.* 655 (2016), 120–134.
[40] Michael W. Mislove. 2000. Nondeterminism and Probabilistic Choice: Obeying the Laws. In *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings*. 350–364.
[41] Michael W. Mislove. 2006. On Combining Probability and Nondeterminism. *Electr. Notes Theor. Comput. Sci.* 162 (2006), 261–265. https://doi.org/10.1016/j.entcs.2005.12.113
[42] Carroll Morgan, Annabelle McIver, and Karen Seidel. 1996. Probabilistic Predicate Transformers. *ACM Trans. Program. Lang. Syst.* 18, 3 (1996), 325–353.
[43] Robert Morris. 1978. Counting Large Numbers of Events in Small Registers. *Commun. ACM* 21, 10 (1978), 840–842.
[44] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *ESOP*. 290–310.
[45] Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. 2008. Hoare type theory, polymorphism and separation. *J. Funct. Program.* 18, 5-6 (2008), 865–911. https://doi.org/10.1017/S0956796808006953
[46] P.W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *TCS* 375, 1 (2007), 271–307.
[47] Adam Petcher and Greg Morrisett. 2015. The Foundational Cryptography Framework. In *POST*. 53–72.
[48] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (1990), 668–676.
[49] Lyle Harold Ramshaw. 1979. *Formalizing the Analysis of Algorithms.* Ph.D. Dissertation. Stanford University.
[50] John C. Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *LICS*.
[51] Wouter Swierstra. 2009. A Hoare Logic for the State Monad. In *TPHOLs*. 440–451.
[52] Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *ESOP*. 909–936.
[53] Iris Team. 2017. Iris 3.0 Documentation. http://plv.mpi-sws.org/iris/appendix-3.0.pdf
[54] Regina Tix, Klaus Keimel, and Gordon D. Plotkin. 2009. Semantic Domains for Combining Probability and Non-Determinism. *Electr. Notes Theor. Comput. Sci.* 222 (2009), 3–99.
[55] Jean-Baptiste Tristan, Joseph Tassarotti, and Guy L. Steele Jr. 2015. Efficient Training of LDA on a GPU by Mean-for-Mode Estimation. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. 59–68.
[56] Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*. 377–390.
[57] V. Vafeiadis and M. Parkinson. 2007. A marriage of rely/guarantee and separation logic. In *CONCUR*. 256–271.
[58] Eelis van der Weegen and James McKinna. 2008. A Machine-Checked Proof of the Average-Case Complexity of Quicksort in Coq. In *TYPES*. 256–271.
[59] Daniele Varacca. 2002. The Powerdomain of Indexed Valuations. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. 299.
[60] Daniele Varacca and Glynn Winskel. 2006. Distributing probability over non-determinism. *Mathematical Structures in Computer Science* 16, 1 (2006), 87–113.

A APPENDIX

In this appendix, we describe in more detail how we modify the definition of Hoare triples in Iris in order to support our probabilistic extensions. Recall from the description in the paper that the main idea is to augment the definition of Hoare triples so that a derivation of a Hoare triple will encode a coupling between each step of the concrete program e and the monadic specification.

$$\mathbb{I} \approx \mathbb{I} \qquad \frac{\mathbb{I}_1 \equiv \mathbb{I}_1' \qquad \mathbb{I}_2 \equiv \mathbb{I}_2' \qquad \mathbb{I}_1 \approx \mathbb{I}_2}{\mathbb{I}_1' \approx \mathbb{I}_2'} \qquad \frac{\mathbb{I}_1 \approx \mathbb{I}_2 \qquad \mathbb{I}_2 \approx \mathbb{I}_3}{\mathbb{I}_1 \approx \mathbb{I}_3}$$

$$\frac{\mathbb{I}_1 \approx \mathbb{I}_2 \qquad \forall x.\, F_1(x) \approx F_2(x)}{x \leftarrow \mathbb{I}_1 ;\ F_1(x) \approx x \leftarrow \mathbb{I}_2 ;\ F_2(x)} \qquad (x \leftarrow \mathbb{I}_1 ;\ \mathbb{I}_2) \approx \mathbb{I}_2$$

Fig. 16. Rules for equivalence up to irrelevant choices

$$\mathcal{I} \lessapprox \mathcal{I} \qquad \frac{\mathcal{I}_1 \lessapprox \mathcal{I}_2 \qquad \mathcal{I}_2 \lessapprox \mathcal{I}_3}{\mathcal{I}1 \lessapprox \mathcal{I}_3} \qquad \frac{\mathcal{I}_1' \subseteq \mathcal{I}_1 \qquad \mathcal{I}_2 \subseteq \mathcal{I}_2' \qquad \mathcal{I}_1 \lessapprox \mathcal{I}_2}{\mathcal{I}_1' \lessapprox \mathcal{I}_2'}$$

$$\frac{\mathcal{I}_1 \lessapprox \mathcal{I}_2 \qquad \forall x.\, F_1(x) \lessapprox F_2(x)}{x \leftarrow \mathcal{I}_1 ;\ F_1(x) \lessapprox x \leftarrow \mathcal{I}_2 ;\ F_2(x)} \qquad (x \leftarrow \mathcal{I}_1 ;\ \mathcal{I}_2) \lessapprox \mathcal{I}_2$$

Fig. 17. Rules for $\lessapprox$ relation.

As we alluded to in the body of the paper, one complexity that arises is that we want to support rules like Ht-NonCouple where no coupling is exhibited in the premise. In these cases, we would like to show that there is a "dummy" coupling we can insert.

Couplings up to Irrelevance. Unfortunately, the analogy between the classical theory of couplings and the indexed valuation formulation breaks down in one key respect that complicates this. In the classical definition, there is *always* at least a trivial coupling between two distributions. Given A and B, we can take the coupling C to be the distribution:

$$x \leftarrow A;\ y \leftarrow B;\ \mathsf{ret}\ (x, y)$$

One can show that this satisfies the two coupling conditions. However, the proof relies on the fact that in the probability distribution monad, $(x \leftarrow A;\ D) \equiv D$ if x does not appear free in D, which in turn is connected to the fact that $A \oplus_p A \equiv A$. As we have noted, an essential fact about the indexed valuation monad is that this equivalence does *not* hold there.

To deal with the fact that there is not always a coupling between indexed valuations, we start by introducing a coarser notion of equivalence between two indexed valuations. The relation $\mathbb{I}_1 \approx \mathbb{I}_2$, which we call equivalence up to irrelevant choices, is inductively defined by the rules in Figure 16. The key rule says $x \leftarrow \mathbb{I}_1 ;\ \mathbb{I}_2 \approx \mathbb{I}_2$, which captures the idea that if the outcome of $\mathbb{I}_1$ is not used subseqently, the computation is equivalent to a version in which $\mathbb{I}_1$ is never executed. We can similarly define an ordering $\mathcal{I}_1 \lessapprox \mathcal{I}_2$, which extends the $\subseteq$ ordering up to irrelevant choices; the rules defining this relation are given in Figure 17.

Definition A.1. We say $\mathbb{I} \approx \mathcal{I} : P$ if there exists $\mathbb{I}'$ and $\mathcal{I}'$ such that: (1) $\mathbb{I} \approx \mathbb{I}'$; (2) $\mathcal{I}' \lessapprox \mathcal{I}$; and (3) $\mathbb{I}' \sim \mathcal{I}' : P$. The latter coupling is called the *witness.*

We call this a "P-coupling up to irrelevance". The following theorem is an analogue of Theorem 2.1 which shows that the existence of this weaker kind of coupling is still sufficient to relate the expected value of $\mathbb{I}$ to that of $\mathcal{I}$:

$$\textsc{Irrel-Equiv}\quad \frac{\mathbb{I} \approx \mathbb{I}' \qquad \mathcal{I} \lessapprox \mathcal{I}' \qquad \mathbb{I} \approx \mathcal{I} : P}{\mathbb{I}' \approx \mathcal{I}' : P} \qquad\qquad \textsc{Irrel-Conseq}\quad \frac{\mathbb{I} \sim \mathcal{I} : P \qquad \forall x, y.\, P(x,y) \rightarrow P'(x,y)}{\mathbb{I} \approx \mathcal{I} : P'}$$

$$\textsc{Irrel-Bind}\quad \frac{\mathbb{I} \approx \mathcal{I} : P \qquad \forall x, y.\, P(x,y) \rightarrow F(x) \sim F'(y) : Q}{(x \leftarrow \mathbb{I};\ F(x)) \approx (y \leftarrow \mathcal{I};\ F'(y)) : Q} \qquad \textsc{Irrel-Regular}\quad \frac{\mathbb{I} \sim \mathcal{I} : P}{\mathbb{I} \approx \mathcal{I} : P} \qquad \textsc{Irrel-Trivial}\quad \mathbb{I} \approx \mathcal{I} : \mathsf{True}$$

$$\textsc{Irrel-Support}\quad \frac{\mathbb{I} \approx \mathcal{I} : P}{\mathbb{I} \approx \mathcal{I} : (\lambda x, y. P(x,y) \wedge x \in \mathsf{supp}(\mathbb{I}) \wedge y \in \mathsf{supp}(\mathcal{I}))}$$

Fig. 18. Rules for constructing couplings up to irrelevance.

Theorem A.2. *For all f and g, if $P(x,y) = (f(x) = g(y))$, and g is bounded on the support of $\mathcal{I}$, then $\mathbb{I} \approx \mathcal{I} : P$ implies that $\mathbb{E}_f[\mathbb{I}]$ exists and:*

$$\mathbb{E}_g^{min}[\mathcal{I}] \leq \mathbb{E}_f[\mathbb{I}] \leq \mathbb{E}_g^{max}[\mathcal{I}]$$

Proof. If $\mathbb{I} \approx \mathbb{I}'$, then $\mathbb{E}_f[\mathbb{I}] = \mathbb{E}_f[\mathbb{I}']$, and if $\mathcal{I}' \lessapprox \mathcal{I}$:

$$\mathbb{E}_g^{\mathsf{min}}[\mathcal{I}] \leq \mathbb{E}_g^{\mathsf{min}}[\mathcal{I}'] \qquad\qquad \mathbb{E}_g^{\mathsf{max}}[\mathcal{I}'] \leq \mathbb{E}_g^{\mathsf{max}}[\mathcal{I}]$$

In addition, the support of $\mathcal{I}'$ is a subset of that of $\mathcal{I}$, so g is bounded on the support of $\mathcal{I}'$. The desired bound then follows from Theorem 2.1. □

Rules for constructing these couplings are shown in Figure 18. Irrel-Regular lets us derive a coupling up to irrelevance from a non-deterministic coupling. Irrel-Trivial shows there is always a coupling up to irrelevance between any $\mathbb{I}$ and $\mathcal{I}$. Then Irrel-Support says we can strengthen the post-condition of a coupling to assume that the returned values are in the supports of $\mathbb{I}$ and $\mathcal{I}$.

Suppose we have $\mathbb{I} : M_\mathsf{I}(T_1)$ and $\mathcal{I} : M_\mathsf{N}(M_\mathsf{I}(T_2))$. Remember that a non-deterministic coupling $\mathbb{I} \sim \mathcal{I} : R$ consists of an $\mathbb{I}' \in \mathcal{I}$, and an R-coupling between $\mathbb{I}$ and $\mathbb{I}'$. Since this latter coupling is itself just some indexed valuation $C : M_\mathsf{I}(T_1 \times T_2)$, we can meaningfully talk about its support and image. Given $x \in T_1$, we say $y \in \mathsf{rsupp}(C, x)$ if $(x, y) \in \mathsf{supp}(C)$. We will simply write $C : \mathbb{I} \sim \mathcal{I} : R$ to refer to this underlying coupling. Similarly, since the witness for a coupling up to irrelevance is a non-deterministic coupling, we will write $C : \mathbb{I} \approx \mathcal{I} : R$ to denote the witness.

Modifying Weakest-Precondition. From here on, we assume familiarity with the Iris model.

Let *ProbState* be the type $\Sigma_{T:Type}\, M_\mathsf{N}(M_\mathsf{I}(T))$. Given two terms $(T_1, \mathcal{I}_1)$ and $(T_2, \mathcal{I}_2)$ of type *ProbState*, we say $(T_1, \mathcal{I}_1) \equiv (T_2, \mathcal{I}_2)$ if $T_1 = T_2$ and $\mathcal{I}_1 \equiv \mathcal{I}_2$. Using this equivalence relation, we impose a discrete OFE structure on *ProbState*. Here, we will often simply omit the type T when writing an element of *ProbState*.

The monad specification resource is handled much in the same way that physical state is in Iris. We represent the monoid specification code using the "authoritative exclusive resource" construction. Define:

$$\mathsf{PInterp}(\mathcal{I}) \triangleq \bullet\, \mathsf{Ex}\ \mathcal{I} \qquad\qquad \mathsf{Prob}(\mathcal{I}) \triangleq \exists \mathcal{I}'.\, \mathcal{I} \subseteq \mathcal{I}' * \circ\, \mathsf{Ex}\ \mathcal{I}'$$

In Iris, Hoare triples are defined in terms of weakest-preconditions, so we actually need to modify the definition of the latter. Recall the following definition of the weakest-precondition, which is taken from the Iris 3.0 documentation [53] (with a typo corrected):

$$
\begin{aligned}
wp \triangleq \mu wp.\, \lambda \mathcal{E}, e, \varphi.\\
&(\exists v.\, \mathsf{expr2val}(e) = v \land \Rrightarrow_{\mathcal{E}} \varphi(v)) \lor \\
&\Big(\mathsf{expr2val}(e) = \bot \land \forall \sigma.\, I(\sigma) \;{}^{\mathcal{E}}\!\Rrightarrow^{\emptyset} \\
&\qquad \mathsf{red}(e, \sigma) * \triangleright \forall e', \sigma', \vec{e}.\, (e, \sigma \to e', \sigma', \vec{e}) \;{}^{\emptyset}\!\Rrightarrow^{\mathcal{E}} \\
&\qquad\qquad I(\sigma') * wp(\mathcal{E}, e', \varphi) * \mathop{\Huge *}_{e'' \in \vec{e}} wp(\top, e'', \lambda\_.\, \mathsf{True})\Big)
\end{aligned}
$$

The definition is a guarded fixed-point, composed of a disjunction which handles two cases: (1) either the expression e is value, in which case the post-condition φ should hold for that value, or (2) it is not a value, in which case for each possible state σ, given the interpretation of σ, we need to show e is reducible, and then recursively show that for each thing which e could step to, we will be able to update the state interpretation appropriately and recursively prove weakest-precondition for the reduct.

We write $\mathsf{primStep}(e, \sigma)$ for the indexed valuation of type $Option(Expr \times State \times List\ Expr)$ which returns reducts of $e; \sigma$ or None if $e; \sigma$ is not reducible.

In our version, the definition of weakest precondition becomes:

$$
\begin{aligned}
wp \triangleq \mu wp.\, \lambda \mathcal{E}, e, \varphi.\\
&(\exists v.\, \mathsf{expr2val}(e) = v \land \Rrightarrow_{\mathcal{E}} \varphi(v)) \lor \\
&\Big(\mathsf{expr2val}(e) = \bot \land \forall \sigma, \mathcal{I}.\, I(\sigma) * \mathsf{PInterp}(\mathcal{I}) \;{}^{\mathcal{E}}\!\Rrightarrow^{\emptyset} \\
&\qquad \mathsf{red}(e, \sigma) * \triangleright \exists R, \mathcal{I}', F, (C : \mathsf{primStep}(e, \sigma) \approx \mathcal{I}' : R), \\
&\qquad\quad (x \leftarrow \mathcal{I}';\ F(x) \subseteq \mathcal{I}) \land \forall e', \sigma', \vec{e}, v'. \\
&\qquad\quad (v' \in \mathsf{rsupp}(C, (e', \sigma', \vec{e})) \land (e, \sigma \to e', \sigma', \vec{e})) \\
&\qquad\quad {}^{\emptyset}\!\Rrightarrow^{\mathcal{E}} I(\sigma') * \mathsf{PInterp}(F(v')) * wp(\mathcal{E}, e', \varphi) \\
&\qquad\qquad * \mathop{\Huge *}_{e'' \in \vec{e}} wp(\top, e'', \lambda\_.\, \mathsf{True})\Big)
\end{aligned}
$$

The value case is the same as in the original definition. In the non-value case, in addition to quantifying over the physical state (σ) we also quantify over the probabilistic state ($\mathcal{I}$). Given the interpretations of both forms of state, we need to show not only is $e; \sigma$ reducible, but also, we must show that $x \leftarrow \mathcal{I}';\ F(x) \subseteq \mathcal{I}$ for some $\mathcal{I}'$ and F, and moreover we must exhibit an R-coupling up to irrelevance between $\mathsf{primStep}(e, \sigma)$ and this $\mathcal{I}'$, for some choice of R. Then, for each thing $e; \sigma$ could reduce to, and for each v' in the corresponding rsupp of the coupling, we have to update the physical state appropriately and the probabilistic state to $F(v')$, and recursively prove the weakest precondition for the reduct.